Demosthenes Koptsis
Author & Developer

Getting Started with FLTK & FreeBASIC

Angelo Rosina
Developer

Getting started with FLTK and FreeBASIC

Table of Contents

Introduction

FreeBASIC is a free, BASIC compiler for Windows (32-bit and 64-bit), 32 bit protected-mode DOS (COFF executables, like DJGPP), and Linux (x86, x86_64, and ARM).

It is open source and licensed under the GPL. It is designed to be syntax compatible with QuickBASIC, while expanding on the language and capabilities.

It can create programs for MS-Windows, DOS and Linux, and is being ported to other platforms.

See About FreeBASIC and Main Features.

FLTK is a C++ Graphical User Interface ("GUI") toolkit for UNIX, Microsoft Windows and Apple OS X.

This document is a description of FLTK headers for FreeBASIC written by Angelo Rosini.

Author of this document is Demosthenes Koptsis and Coders are Angelo Rosini and Demosthenes Koptsis.

Thanks to Angelo Rosini the developer of FLTK FreeBASIC headers and all the people of FreeBASIC community who helped with the info and code for this book.

Installation

In this document, we assume that you have the latest version of the FreeBASIC programming language installed on your computer. Also that you have an editor or IDE installed.

We assume that you have not installed any FLTK libraries or FLTK Freebasic headers on your system.

Installation on Linux

1. First you have to download the FLTK tarball from https://www.fltk.org/. The latest stable FLTK package is 1.3.8. You can download it from https://www.fltk.org/software.php. Download the latest source gz or bz tarball on your system.
2. Untar the package and you should have a folder named fltk-1.3.8.
3. Enter the fltk-1.3.8 folder and open a terminal window from it.
4. Read the README text file in order to see what parameters you can pass to configure command.

5. Run the following commands in your terminal in order to compile the FLTK libs.

   ./configure

   make

6. After a successful compilation you should have in your lib folder the following files

   libfltk.a

   libfltk_forms.a

   libfltk_gl.a

   libfltk_images.a

7. Copy these files to your Freebasic lib/freebasic/linux-x86_64 folder.

8. Next you should download the Freebasic FLTK headers from https://github.com/angros47/FLTK-headers-for-FreeBasic

9. Angelo Rosini aka angros47, has written the FLTK headers files for Freebasic. You should download the latest version from github as a zip file.

10. Unzip the FLTK-headers-for-FreeBasic-main.zip file to a folder named FLTK in your Freebasic include/freebasic folder.

    FLTK libs and Freebasic headers are now installed successfully on your system and you are ready to develop FLTK applications with Freebasic.

11. Write an example code

```
1.  #include once "FLTK/Fl_Window.bi"
2.  #include once "FLTK/Fl_Button.bi"
3.
4.  dim w as Fl_Window = Fl_Window(940,380,"Window")
5.        dim b as Fl_button = Fl_button(10,30,150,30,"This is a button")
6.  w.end_()
7.
8.  w.show
9.
10. fl.run_
```

12. You should pass the following parameters to your fbc compiler

```
-l Xrender -l Xcursor -l Xfixes -l Xext -l Xft -l fontconfig -l Xinerama -l pthread
-l m -l X11 -l png -l z -l jpeg
```

13. Compile and Run your bas file

How to compile and run FLTK applications in x86_64 Windows, like Win11

1. First you have to install Msys2 (https://www.msys2.org/)
2. Install for Msys2 MINGW64 the mingw-w64-x86_64-fltk package (https://packages.msys2.org/package/mingw-w64-x86_64-fltk?repo=mingw64)
3. Copy fltk libs to Freebasic lib\win64 folder (FreeBASIC-1.10.0-win64\lib\win64)

   C:\msys64\mingw64\bin\libfltk.dll

   C:\msys64\mingw64\bin\libfltk_forms.dll

   C:\msys64\mingw64\bin\libfltk_gl.dll

   C:\msys64\mingw64\bin\libfltk_images.dll

   C:\msys64\mingw64\lib\libfltk.dll.a

   C:\msys64\mingw64\lib\libfltk_forms.dll.a

   C:\msys64\mingw64\lib\libfltk_gl.dll.a

   C:\msys64\mingw64\lib\libfltk_images.dll.a
4. Create in your favorite freebasic ide (My is Poseidon) the fltk project
5. Add to your project properties the include and lib\win64 folders path
6. Download the FLTK angros47 headers from github (https://github.com/angros47/FLTK-headers-for-FreeBasic)
7. unzip angros47 FLTK headers to Freebasic include folder in a folder named FLTK
8. copy the next files from Msys2 installation folder, mine is C:\msys64\mingw64\bin to your Freebasic Project folder

   C:\msys64\mingw64\bin\libfltk.dll

   C:\msys64\mingw64\bin\libfltk_forms.dll

   C:\msys64\mingw64\bin\libfltk_gl.dll

   C:\msys64\mingw64\bin\libfltk_images.dll

   C:\msys64\mingw64\lib\libfltk.dll.a

   C:\msys64\mingw64\lib\libfltk_forms.dll.a

   C:\msys64\mingw64\lib\libfltk_gl.dll.a

   C:\msys64\mingw64\lib\libfltk_images.dll.a

   C:\msys64\mingw64\bin\libgcc_s_seh-1.dll

   C:\msys64\mingw64\bin\libstdc++-6.dll

C:\msys64\mingw64\bin\libwinpthread-1.dll

9. write an example main.bas file

```
#include once "FLTK/Fl_Window.bi"
#include once "FLTK/Fl_Button.bi"

dim w as Fl_Window = Fl_Window(940,380,"Window")
      dim b as Fl_button = Fl_button(10,30,150,30,"This is a button")
w.end_()

w.show

fl.run_
```

10. Build project
11. Run project

You should have successfully run your first FLTK app in Win11

How to compile and run FLTK applications in Win11 or fbc32

1. First you have to install Msys2 (https://www.msys2.org/)
2. Install for Msys2 MINGW32 the mingw-w64-i686-fltk package (https://packages.msys2.org/package/mingw-w64-i686-fltk?repo=mingw32)
3. Copy fltk libs to Freebasic lib\win32 folder (FreeBASIC-1.10.0-win32\lib\win32)

   C:\msys64\mingw32\bin\libfltk.dll

   C:\msys64\mingw32\bin\libfltk_forms.dll

   C:\msys64\mingw32\bin\libfltk_gl.dll

   C:\msys64\mingw32\bin\libfltk_images.dll

   C:\msys64\mingw32\lib\libfltk.dll.a

   C:\msys64\mingw32\lib\libfltk_forms.dll.a

   C:\msys64\mingw32\lib\libfltk_gl.dll.a

   C:\msys64\mingw32\lib\libfltk_images.dll.a
4. Create in your favorite freebasic ide (My is Poseidon) the fltk project
5. Add to your project properties the include and lib\win32 folders path
6. Download the FLTK angros47 headers from github (https://github.com/angros47/FLTK-headers-for-FreeBasic)
7. unzip angros47 FLTK headers to Freebasic include folder in a folder named FLTK
8. copy the next files from Msys2 installation folder, mine is C:\msys64\mingw32\bin to your Freebasic Project folder

C:\msys64\mingw32\bin\libfltk.dll

C:\msys64\mingw32\bin\libfltk_forms.dll

C:\msys64\mingw32\bin\libfltk_gl.dll

C:\msys64\mingw32\bin\libfltk_images.dll

C:\msys64\mingw32\lib\libfltk.dll.a

C:\msys64\mingw32\lib\libfltk_forms.dll.a

C:\msys64\mingw32\lib\libfltk_gl.dll.a

C:\msys64\mingw32\lib\libfltk_images.dll.a

C:\msys64\mingw32\bin\libgcc_s_dw2-1.dll

C:\msys64\mingw32\bin\libstdc++-6.dll

C:\msys64\mingw32\bin\libwinpthread-1.dll

9. write an example main.bas file

```
#include once "FLTK/Fl_Window.bi"
#include once "FLTK/Fl_Button.bi"

dim w as Fl_Window = Fl_Window(940,380,"Window")
      dim b as Fl_button = Fl_button(10,30,150,30,"This is a button")
w.end_()

w.show

fl.run_
```

10. Build project
11. Run project

You should have successfully run your first FLTK application in Win11 for fbc32

FLTK Basics

Writing Your First FLTK Program

The program must include a header file for each FLTK UDT it uses.

Listing 1 shows a simple "Hello, World!" program that uses FLTK to display the window.

Listing 1 – Project: 001-hello – hello.bas

```
1.  #include "FLTK/Fl_Window.bi"
2.  #include "FLTK/Fl_Box.bi"
3.
4.  Dim fWindow As Fl_Window Ptr
5.  Dim fBox As Fl_Box Ptr
6.  Dim As ZString * 13 str1 => "hello, world"
7.
8.  fWindow  = New Fl_Window(0, 0, 340, 180, str1)
9.        fBox = New Fl_Box(20, 40, 300, 100, str1)
10.       fBox->box(FL_UP_BOX)
11.       fBox->labelfont(FL_BOLD + FL_ITALIC)
12.       fBox->labelsize(36)
13.       fBox->labeltype(FL_SHADOW_LABEL)
14. fWindow->end_()
15. fWindow->show()
16.
17. Fl.run_()
```

After including the required header files, we define two pointer variables one as FL_Window and another as FL_Box.

```
Dim fWindow As Fl_Window Ptr
Dim fBox As Fl_Box Ptr
```

Next we define a Zstring string variable which holds the title of the window.

All following widgets will automatically be children of this window.

```
fWindow  = New Fl_Window(0, 0, 340, 180, str1)
```

Then we create a box with the "Hello, World!" string in it.
FLTK automatically adds the new box to window, the current grouping widget.

```
fBox = New Fl_Box(20, 40, 300, 100, str1)
```

Next, we set the type of box and the font, size, and style of the label:

```
      fBox->box(FL_UP_BOX)
      fBox->labelfont(FL_BOLD + FL_ITALIC)
      fBox->labelsize(36)
      fBox->labeltype(FL_SHADOW_LABEL)
```

We tell FLTK that we will not add any more widgets to window.

```
fWindow->end_()
```

Finally, we show the window and enter the FLTK event loop:

```
fWindow->show()
Fl.run_()
```

The resulting program will display the window in Figure 1.

You can quit the program by closing the window or pressing the ESCape key.

Figure 1

In this example we used pointers.
Although, the next code example is equivalent with the example with pointers.
Personally, I found the second example easier.

```
1.  #include "FLTK_main/Fl_Window.bi"
2.  #include "FLTK_main/Fl_Box.bi"
3.
4.  Dim As ZString * 13 str1 => "hello, world"
5.  Dim fWindow as Fl_Window = Fl_Window(0, 0, 340, 180, str1)
6.        Dim fBox as Fl_Box = Fl_Box(20, 40, 300, 100, str1)
7.        fBox.box(FL_UP_BOX)
8.        fBox.labelfont(FL_BOLD + FL_ITALIC)
9.        fBox.labelsize(36)
10.       fBox.labeltype(FL_SHADOW_LABEL)
11. fWindow.end_()
12.
13. fWindow.show
14.
15. Fl.run_()
```

Creating the Widgets

The widgets are created using the new operator. For most widgets the arguments to the constructor are:
`Fl_Widget(x as long, y as long, w as long, h as long, title as const zstring ptr)`

The x and y parameters determine where the widget or window is placed on the screen.

In FLTK the top left corner of the window or screen is the origin (i.e. x = 0, y = 0) and the units are in pixels.

The w (width) and h (height) parameters determine the size of the widget or window in pixels.
The maximum widget size is typically governed by the underlying window system or hardware.

title is a pointer to a zstring string to label the widget with or NULL.
If not specified the title defaults to NULL.

The label string must be in static storage such as a string constant because FLTK does not make a copy of it - it just uses the pointer.

Creating Widget hierarchies

Widgets are commonly ordered into functional groups, which in turn may be grouped again, creating a hierarchy of widgets.

FLTK makes it easy to fill groups by automatically adding all widgets that are created between a `myGroup->begin()` and `myGroup->end_()`.

In this example, myGroup would be the current group.
Newly created groups and their derived widgets implicitly call begin() in the constructor, effectively adding all subsequently created widgets to itself until end_() is called.

Setting the current group to NULL will stop automatic hierarchies.
New widgets can now be added manually using `myGroup->add(...)` and `myGroup->insert(...)`.

Get/Set Methods

`box->box(FL_UP_BOX)` sets the type of box the Fl_Box draws, changing it from the default of FL_NO_BOX, which means that no box is drawn.

In our "Hello, World!" example we use FL_UP_BOX, which means that a raised button border will be drawn around the widget.

You could examine the boxtype in by doing `box->box()`.

FLTK uses method name overloading to make short names for get/set methods. A "set" method is generally of the form `"declare sub name(parameter...)"`, and a "get" method is generally of the form `"declare const function name()"`.

Redrawing After Changing Attributes

Almost all of the set/get pairs are very fast. However, the "set" methods do not call `redraw()` - you have to call it yourself. This greatly reduces code size and execution time. The only common exceptions are `value()` which calls `redraw()` and `label()` which calls `redraw_label()` if necessary.

Labels

All widgets support labels. In the case of window widgets, the label is used for the label in the title bar.
Our example program calls the `labelfont()`, `labelsize()`, and `labeltype()` methods.
The `labelfont()` method sets the typeface and style that is used for the label, which for this example we are using `FL_BOLD` and `FL_ITALIC`. You can also specify typefaces directly.
The `labelsize()` method sets the height of the font in pixels.
The `labeltype()` method sets the type of label.
FLTK supports normal, embossed, and shadowed labels internally, and more types can be added as desired.

Showing the Window

The `show()` method shows the widget or window.

The Main Event Loop

All FLTK applications (and most GUI applications in general) are based on a simple event processing model. User actions such as mouse movement, button clicks, and keyboard activity generate events that are sent to an application. The application may then ignore the events or respond to the user, typically by redrawing a button in the "down" position, adding the text to an input field, and so forth.

FLTK also supports idle, timer, and file pseudo-events that cause a function to be called when they occur. Idle functions are called when no user input is present and no timers or files need to be handled in short, when the application is not doing anything. Idle callbacks are often used to update a 3D display or do other background processing.

Timer functions are called after a specific amount of time has expired. They can be used to pop up a progress dialog after a certain amount of time or do other things that need to happen at more-or-less regular intervals. FLTK timers are not 100% accurate, so they should not be used to measure time intervals, for example.

File functions are called when data is ready to read or write, or when an error condition occurs on a file. They are most often used to monitor network connections (sockets) for data-driven displays.

FLTK applications must periodically check (`Fl.check()`) or wait (`Fl.wait()`) for events or use the `Fl.run()` method to enter a standard event processing loop. Calling `Fl.run()` is equivalent to the following code:

```
while (Fl.wait())
```

`Fl.run()` does not return until all of the windows under FLTK control are closed by the user or your program.

Naming

All public symbols in FLTK start with the characters 'F' and 'L':

- Functions are either Fl.foo().
- UDT and type names are capitalized: Fl_Foo.
- Constants and enumerations are uppercase: FL_FOO.
- All header files start with “FLTK/...”.

Header Files

The proper way to include FLTK header files is:
#include “FLTK/Fl_xyz.bi”

Note
Case is significant on many operating systems, and the C standard uses the forward slash (/) to separate directories. Do not use any of the following include lines:
#include “FLTK\Fl_xyz.bi”
#include “fltk/fl_xyz.bi”
#include “FlTK/fl_xyz.bi”

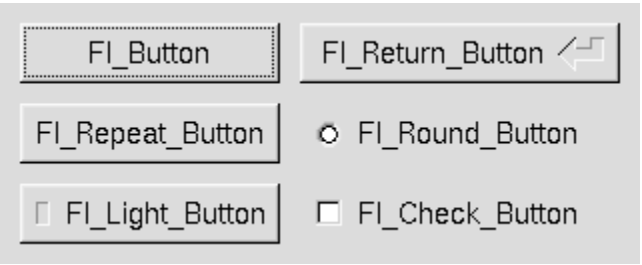
Common Widgets and Attributes

This chapter describes many of the widgets that are provided with FLTK and covers how to query and set the standard attributes.

Buttons

FLTK provides many types of buttons:
• Fl_Button - A standard push button.
• Fl_Check_Button - A button with a check box.
• Fl_Light_Button - A push button with a light.
• Fl_Repeat_Button - A push button that repeats when held.
• Fl_Return_Button - A push button that is activated by the Enter key.
• Fl_Round_Button - A button with a radio circle.

Figure 2



All of these buttons just need the corresponding “FLTK/Fl_xyz_Button.bi” header file.
The constructor takes the bounding box of the button and optionally a label string:

```
Dim button as Fl_button = Fl_button(x, y, width, height, "label")
Dim lbutton as Fl_Light_Button = Fl_Light_Button(x, y, width, height)
Dim rbutton as Fl_Round_Button = Fl_Round_Button(x, y, width, height, "label")
```

Each button has an associated `type()` which allows it to behave as a push button, toggle button, or radio button:

```
button.type_(FL_NORMAL_BUTTON)
lbutton.type_(FL_TOGGLE_BUTTON)
rbutton.type_(FL_RADIO_BUTTON)
```

see Project 002-buttons – buttons.bas

```
1.  #include once "FLTK/Fl_Window.bi"
2.  #include once "FLTK/Fl_Button.bi"
3.  #include once "FLTK/Fl_Light_Button.bi"
4.  #include once "FLTK/Fl_Round_Button.bi"
5.
6.  Dim w as Fl_Window = Fl_Window(200,200,"Window")
7.    Dim button as Fl_button = Fl_button(10, 10, 100, 20, "FL_Button")
8.    Dim lbutton as Fl_Light_Button = Fl_Light_Button(10, 50, 130, 20,
    "FL_Light_Button")
9.    Dim rbutton as Fl_Round_Button = Fl_Round_Button(10, 90, 100, 20,
    "FL_Round_Button")
10.
11.   button.type_(FL_NORMAL_BUTTON)
12.   lbutton.type_(FL_TOGGLE_BUTTON)
13.   rbutton.type_(FL_RADIO_BUTTON)
14.
15. w.end_()
16.
17. w.show
18. fl.run_
```

For toggle and radio buttons, the `value()` method returns the current button state (0 = off, 1 = on).

The `set()` and `clear()` methods can be used on toggle buttons to turn a toggle button on or off, respectively. Radio buttons can be turned on with the `setonly()` method; this will also turn off other radio buttons in the same group.

Text

FLTK provides several text widgets for displaying and receiving text:
• Fl_Input - A one-line text input field.
• Fl_Output - A one-line text output field.
• Fl_Multiline_Input - A multi-line text input field.
• Fl_Multiline_Output - A multi-line text output field.
• Fl_Text_Display - A multi-line text display widget.
• Fl_Text_Editor - A multi-line text editing widget.
• Fl_Help_View - A HTML text display widget.

The Fl_Output and Fl_Multiline_Output widgets allow the user to copy text from the output field but not change it.

The `value()` method is used to get or set the string that is displayed:

```
Dim sInput As Fl_Input = Fl_Input(x, y, width, height, "label")
sInput.value("Now is the time for all good men...")
```

The string is copied to the widget's own storage when you set the `value()` of the widget.

The Fl_Text_Display and Fl_Text_Editor widgets use an associated Fl_Text_Buffer UDT for the value, instead of a simple string.

See Project 003-text – text.bas

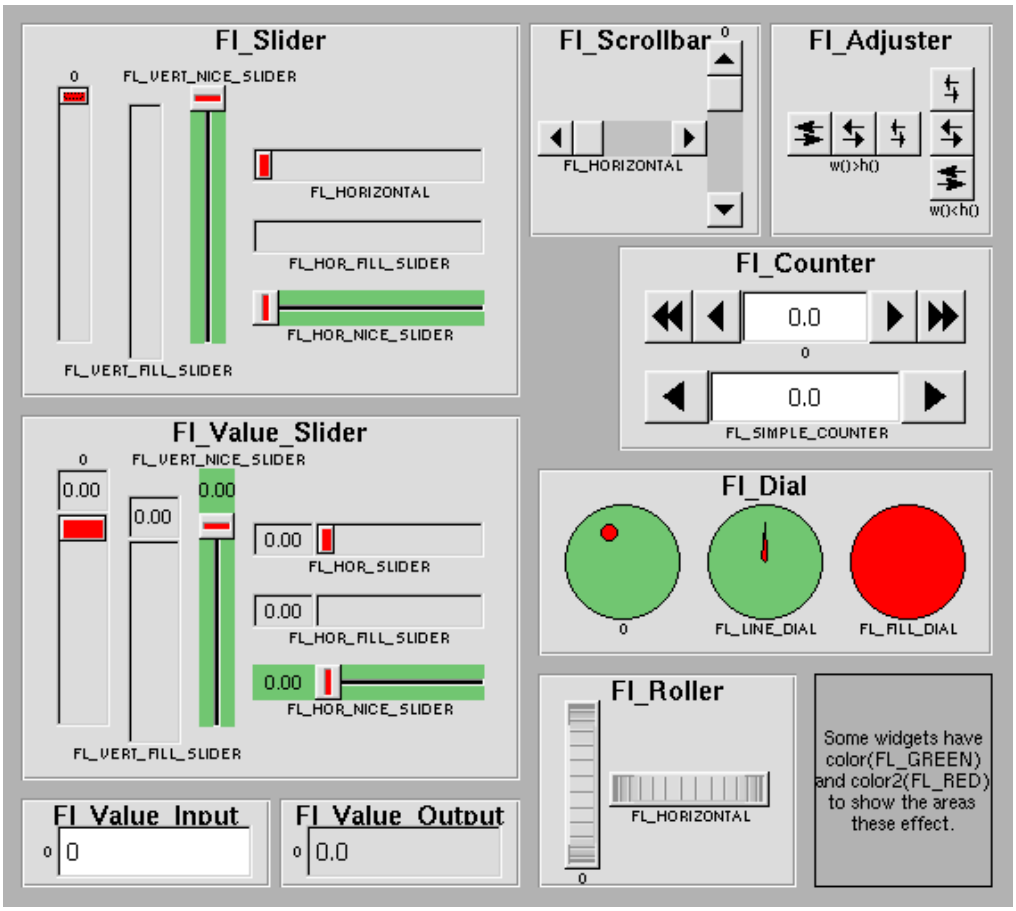
```
#include once "FLTK/Fl_Window.bi"
#include once "FLTK/Fl_Input.bi"

Dim w as Fl_Window = Fl_Window(400,200,"Window")
      Dim sInput As Fl_Input = Fl_Input(50, 50, 250, 20, "label")
      sInput.value("Now is the time for all good men...")
w.end_()

w.show
fl.run_
```

Valuators

Unlike text widgets, valuators keep track of numbers instead of strings.
FLTK provides the following valuators:

- Fl_Counter - A widget with arrow buttons that shows the current value.
- Fl_Dial - A round knob.
- Fl_Roller - An SGI-like dolly widget.
- Fl_Scrollbar - A standard scrollbar widget.
- Fl_Slider - A scrollbar with a knob.
- Fl_Value_Slider - A slider that shows the current value.

Figure 3



The `value()` method gets and sets the current value of the widget.
The `minimum()` and `maximum()` methods set the range of values that are reported by the widget.

See Project 004-valuator – scrollbar.bas

```
1.  #include once "FLTK/Fl_Window.bi"
2.  #include once "FLTK/Fl_Scrollbar.bi"
3.
4.  Dim w as Fl_Window = Fl_Window(400,200,"Window")
5.       Dim vScrollBar As Fl_Scrollbar=Fl_Scrollbar(10,50,300,24,"Scrollbar")
6.       vScrollBar.type_(FL_HORIZONTAL)
7.       vScrollBar.minimum(0.00)
8.       vScrollBar.maximum(100.00)
9.       vScrollBar.value(50.00)
10. w.end_()
11.
12. w.show
13. fl.run_
```

Groups

The Fl_Group widget UDT is used as a general purpose "container" widget.
Besides grouping radio buttons, the groups are used to encapsulate windows, tabs, and scrolled windows.
The following group classes are available with FLTK:
• Fl_Double_Window - A double-buffered window on the screen.
• Fl_Gl_Window - An OpenGL window on the screen.
• Fl_Group - The base container class; can be used to group any widgets together.
• Fl_Pack - A collection of widgets that are packed into the group area.
• Fl_Scroll - A scrolled window area.
• Fl_Tabs - Displays child widgets as tabs.
• Fl_Tile - A tiled window area.
• Fl_Window - A window on the screen.
• Fl_Wizard - Displays one group of widgets at a time.

Setting the Size and Position of Widgets

The size and position of widgets is usually set when you create them.
You can access them with the x(), y(), w(), and h() methods.
You can change the size and position by using the position(), resize(), and size() methods:

```
button->position(x, y)                  button.position(x, y)
group->resize(x, y, width, height)      group.resize(x, y, width, height)
window->size(width, height)             window.size(width, height)
```

If you change a widget's size or position after it is displayed you will have to call redraw() on the widget's parent.

Colors

FLTK stores the colors of widgets as an 32-bit unsigned number that is either an index into a color palette of 256 colors or a 24-bit RGB color.

The color palette is not the X or MS Windows colormap, but instead is an internal table with fixed contents.

See the Colors section of Drawing Things in FLTK for implementation details.

There are symbols for naming some of the more common colors:
• FL_BLACK
• FL_RED
• FL_GREEN
• FL_YELLOW
• FL_BLUE
• FL_MAGENTA
• FL_CYAN
• FL_WHITE
• FL_WHITE

Other symbols are used as the default colors for all FLTK widgets.
• FL_FOREGROUND_COLOR
• FL_BACKGROUND_COLOR
• FL_INACTIVE_COLOR
• FL_SELECTION_COLOR

The full list of named color values can be found in FLTK Enumerations.

A color value can be created from its RGB components by using the `fl_rgb_color()` function, and decomposed again with `Fl.get_color()`:

```
Fl_Color c = fl_rgb_color(85, 170, 255)
Fl.get_color(c, r, g, b);
// RGB to Fl_Color
// Fl_Color to RGB
```

The widget color is set using the color() method:

```
button->color(FL_RED)
// set color using named value
```

Similarly, the label color is set using the labelcolor() method:

```
button->labelcolor(FL_WHITE)
```

The Fl_Color encoding maps to a 32-bit unsigned integer representing RGBI, so it is also possible to specify a color using a hex constant as a color map index:

```
button->color(&h0000ff)
// colormap index #255 (FL_WHITE)
```

or specify a color using a hex constant for the RGB components:

```
button->color(&hff000000)
button->color(&h00ff0000)
button->color(&h0000ff00)
```

```
button->color(&hffffff00)
// RGB: red
// RGB: green
// RGB: blue
// RGB: white
```

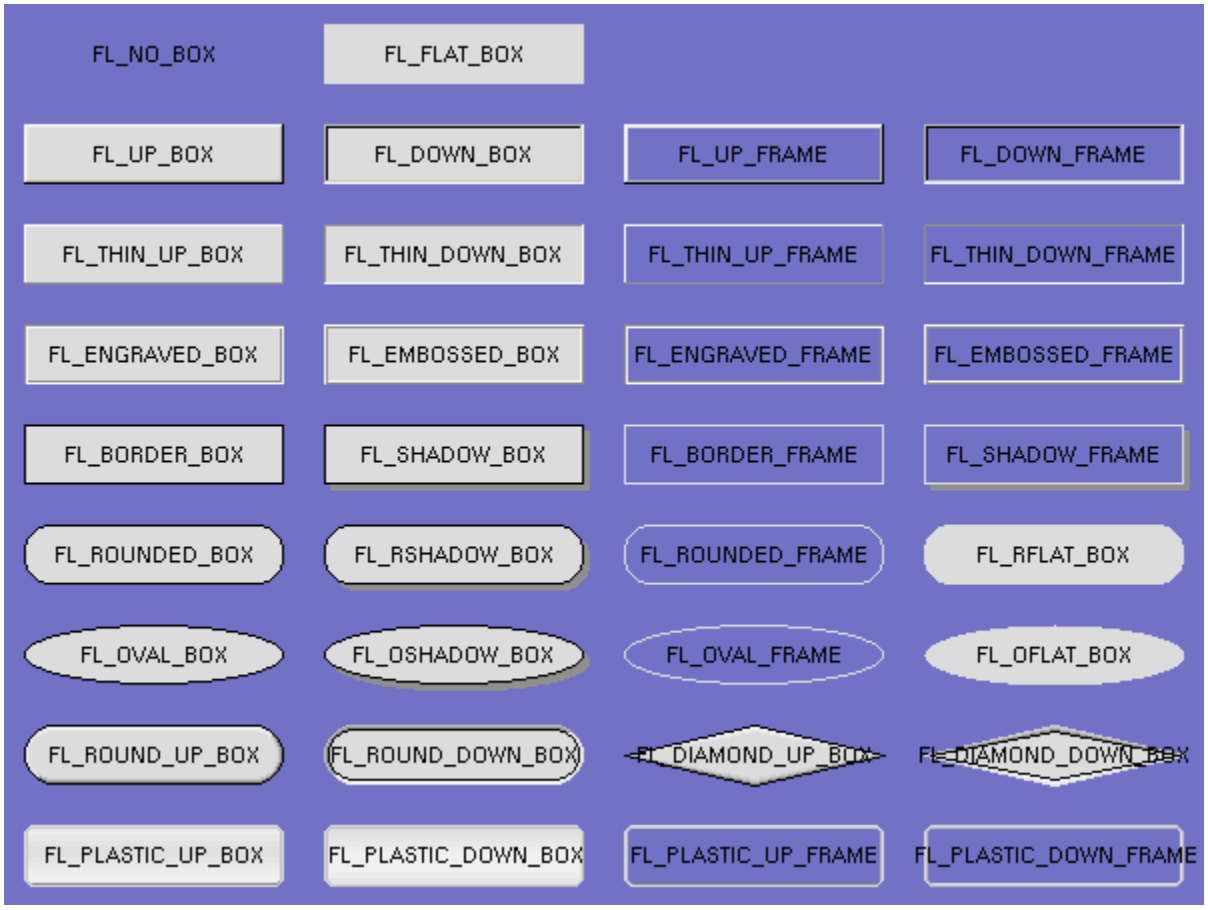
Note
If TrueColor is not available, any RGB colors will be set to the nearest entry in the colormap.

Box Types

The type Fl_Boxtype stored and returned in Fl_Widget.box() is an enumeration defined in Enumerations.bi.

Figure 4 shows the standard box types included with FLTK.

Figure 4



FL_NO_BOX means nothing is drawn at all, so whatever is already on the screen remains.
The FL_..._FRAME types only draw their edges, leaving the interior unchanged.
The blue color in Figure 4 is the area that is not drawn by the frame types.

Labels and Label Types

The `label()`, `align()`, `labelfont()`, `labelsize()`, `labeltype()`, `image()`, and `deimage()` methods control the labeling of widgets.

label()

The `label()` method sets the string that is displayed for the label. Symbols can be included with the label string by escaping them using the "@" symbol - "@@" displays a single at sign. Figure 5 shows the available symbols.

Figure 5

The @ sign may also be followed by the following optional "formatting" characters, in this order:
• '#' forces square scaling, rather than distortion to the widget's shape.
• +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
• '$' flips the symbol horizontally, '%' flips it vertically.
• [0-9] - rotates by a multiple of 45 degrees. '5' and '6' do no rotation while the others point in the direction of that key on a numeric keypad. '0', followed by four more digits rotates the symbol by that amount in degrees.

Thus, to show a very large arrow pointing downward you would use the label string "@+92->"

align()

The `align()` method positions the label. The following constants are defined and may be OR'd together as needed:
• FL_ALIGN_CENTER - center the label in the widget.
• FL_ALIGN_TOP - align the label at the top of the widget.
• FL_ALIGN_BOTTOM - align the label at the bottom of the widget.
• FL_ALIGN_LEFT - align the label to the left of the widget.
• FL_ALIGN_RIGHT - align the label to the right of the widget.
• FL_ALIGN_LEFT_TOP - The label appears to the left of the widget, aligned at the top. Outside labels only.
• FL_ALIGN_RIGHT_TOP - The label appears to the right of the widget, aligned at the top. Outside labels only.
• FL_ALIGN_LEFT_BOTTOM - The label appears to the left of the widget, aligned at the bottom. Outside labels only.
• FL_ALIGN_RIGHT_BOTTOM - The label appears to the right of the widget, aligned at the bottom. Outside labels only.
• FL_ALIGN_INSIDE - align the label inside the widget.
• FL_ALIGN_CLIP - clip the label to the widget's bounding box.
• FL_ALIGN_WRAP - wrap the label text as needed.
• FL_ALIGN_TEXT_OVER_IMAGE - show the label text over the image.
• FL_ALIGN_IMAGE_OVER_TEXT - show the label image over the text (default).
• FL_ALIGN_IMAGE_NEXT_TO_TEXT - The image will appear to the left of the text.
• FL_ALIGN_TEXT_NEXT_TO_IMAGE - The image will appear to the right of the text.
• FL_ALIGN_IMAGE_BACKDROP - The image will be used as a background for the widget.

labeltype()

The `labeltype()` method sets the type of the label. The following standard label types are included:
• FL_NORMAL_LABEL - draws the text.
• FL_NO_LABEL - does nothing.
• FL_SHADOW_LABEL - draws a drop shadow under the text.
• FL_ENGRAVED_LABEL - draws edges as though the text is engraved.
• FL_EMBOSSED_LABEL - draws edges as thought the text is raised.
• FL_ICON_LABEL - draws the icon associated with the text.

image() and deimage()

The `image()` and `deimage()` methods set an image that will be displayed with the widget.
The `deimage()` method sets the image that is shown when the widget is inactive, while the `image()` method sets the image that is shown when the widget is active.

To make an image you use a subclass of Fl_Image.

Callbacks

Callbacks are functions that are called when the value of a widget changes.
A callback function is sent a Fl_Widget pointer of the widget that changed and a pointer to data that you provide:

```
sub xyz_callback(w as Fl_Widget ptr, data_ as any ptr)
...
end sub
```

The callback() method sets the callback function for a widget.
You can optionally pass a pointer to some data needed for the callback:

```
dim xyz_data as long
button->callback(@xyz_callback, @xyz_data)
```

Normally callbacks are performed only when the value of the widget changes.
You can change this using the Fl_Widget.when() method:

```
button->when(FL_WHEN_NEVER)
button->when(FL_WHEN_CHANGED)
button->when(FL_WHEN_RELEASE)
button->when(FL_WHEN_RELEASE_ALWAYS)
button->when(FL_WHEN_ENTER_KEY)
button->when(FL_WHEN_ENTER_KEY_ALWAYS)
button->when(FL_WHEN_CHANGED Or FL_WHEN_NOT_CHANGED)
```

Shortcuts

Shortcuts are key sequences that activate widgets such as buttons or menu items.
The `shortcut()` method sets the shortcut for a widget:

```
button->shortcut(FL_Enter)
button->shortcut(__FL_SHIFT & "b")
button->shortcut(__FL_CTRL + "b")
button->shortcut(__FL_ALT + "b")
button->shortcut(__FL_CTRL + __FL_ALT + "b")
button->shortcut(0) ‘ no shortcut
```

The shortcut value is the key event value - the ASCII value or one of the special keys described in `Fl.event_key()` values combined with any modifiers like Shift , Alt , and Control.

Drawing Things in FLTK

This chapter covers the drawing functions that are provided with FLTK.

When Can You Draw Things in FLTK?

There are only certain places you can execute FLTK code that draws to the computer's display.
Calling these functions at other places will result in undefined behavior!

• The most common place is inside the virtual `Fl_Widget.draw()` method.
To write code here, you must subclass one of the existing Fl_Widget classes and implement your own version of draw().
• You can also create custom boxtypes and labeltypes. These involve writing small procedures that can be called by existing `Fl_Widget.draw()` methods.
These "types" are identified by an 8-bit index that is stored in the widget's box(), labeltype(), and possibly other properties.
• You can call `Fl_Window.make_current()` to do incremental update of a widget.
Use `Fl_Widget.window()` to find the window.

In contrast, code that draws to other drawing surfaces than the display
(i.e., instances of derived classes of the `Fl_Surface_Device` UDT, except `Fl_Display_Device`, such as `Fl_Printe`r and `Fl_Copy_Surface`)
can be executed at any time as follows:

1. Memorize what is the current drawing surface calling `Fl_Surface_Device.surface()`, and make your surface
the new current drawing surface calling the surface's `set_current()` function;
2. Make a series of calls to any of the drawing functions described below; these will operate on the new current drawing surface;
3. Set the current drawing surface back to its previous state calling the previous surface's `set_current()`.

What Drawing Unit do FLTK drawing functions use?

When drawing to the display or to instances of `Fl_Copy_Surface` and `Fl_Image_Surface`, the unit of drawing functions corresponds generally to one pixel.
The so-called 'retina' displays of some recent Apple computers are an exception to this rule:
one drawing unit corresponds to the width or the height of 2 display pixels on a retina display.

When drawing to surfaces that are instances of `Fl_Paged_Device` derived classes (i.e., `Fl_Printer` or `Fl_PostScript_File_Device`), the drawing unit is initially one point, that is, 1/72 of an inch.
But this unit is changed after calls to `Fl_Paged_Device.scale()`.

Drawing Functions

To use the drawing functions you must first include the FLTK/fl_draw.bi header file.
FLTK provides the following types of drawing functions:

• Boxes
• Clipping
• Colors
• Line Dashes and Thickness
• Drawing Fast Shapes
• Drawing Complex Shapes
• Drawing Text
• Fonts
• Character Encoding
• Drawing Overlays
• Drawing Images
• Direct Image Drawing
• Direct Image Reading
• Image Classes
• Offscreen Drawing

Boxes

FLTK provides three functions that can be used to draw boxes for buttons and other UI controls.
Each function uses the supplied upper-lefthand corner and width and height to determine where to draw the box.

```
declare sub fl_draw_box(as Fl_Boxtype, x as long, y as long, w as long, h as long,
as Fl_Color)
```

The `fl_draw_box()` function draws a standard boxtype b in the specified color c.

```
declare sub fl_frame(s as const zstring ptr, x as long, y as long, w as long, h as
long)
declare sub fl_frame2(s as const zstring ptr, x as long, y as long, w as long, h as
long)
```

The `fl_frame()` and `fl_frame2()` functions draw a series of line segments around the given box.

The string s must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white.

The results of calling these functions with a string that is not a multiple of 4 characters in length are undefined.

The only difference between `fl_frame()` and `fl_frame2()` is the order of the line segments:
• For `fl_frame()` the order of each set of 4 characters is: top, left, bottom, right.
• For `fl_frame2()` the order of each set of 4 characters is: bottom, right, top, left.

Note that `fl_frame(Fl_Boxtype b)` is described in the Box Types section.

Clipping

You can limit all your drawing to a rectangular region by calling `fl_push_clip()`, and put the drawings back by using `fl_pop_clip()`.
This rectangle is measured in pixels and is unaffected by the current transformation matrix.

In addition, the system may provide clipping when updating windows which may be more complex than a simple rectangle.

```
private sub fl_push_clip(x as long, y as long, w as long, h as long)
sub fl_clip(int x, int y, int w, int h)
```

Intersect the current clip region with a rectangle and push this new region onto the stack.
The `fl_clip()` version is deprecated and will be removed from future releases.

```
private sub fl_push_no_clip()
```
Pushes an empty clip region on the stack so nothing will be clipped.

```
private sub fl_pop_clip()
```
Restore the previous clip region.

Note: You must call `fl_pop_clip()` once for every time you call `fl_push_clip()`.
If you return to FLTK with the clip stack not empty unpredictable results occur.

```
private function fl_not_clipped(x as long, y as long, w as long, h as long) as long
```
Returns non-zero if any of the rectangle intersects the current clip region.
If this returns 0 you don't have to draw the object.
Note: Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.

```
private function fl_clip_box(x as long, y as long, w as long, h as long, byref x1
as long,  byref y1 as long, byref w1 as long,  byref h1 as long) as long
```
Intersect the rectangle x,y,w,h with the current clip region and returns the bounding box of the result in X,Y,W,H.

Returns non-zero if the resulting rectangle is different than the original. This can be used to limit the necessary drawing to a rectangle.
W and H are set to zero if the rectangle is completely outside the region.

```
private sub fl_clip_region overload(r as Fl_Region)
private function fl_clip_region overload() as Fl_Region
```
Replace the top of the clip stack with a clipping region of any shape. Fl_Region is an operating system specific type. The second form returns the current clipping region.

Colors

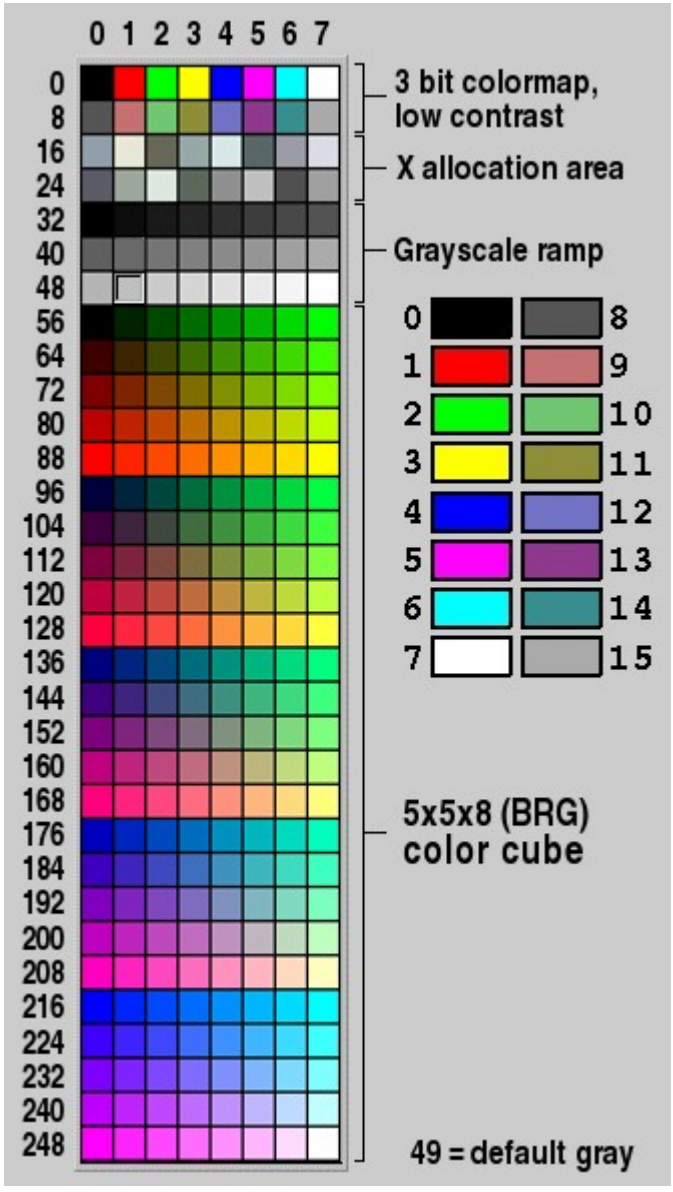
FLTK manages colors as 32-bit unsigned integers, encoded as RGBI. When the "RGB" bytes are non-zero, the value is treated as RGB. If these bytes are zero, the "I" byte will be used as an index into the colormap.

Colors with both "RGB" set and an "I" >0 are reserved for special use.
Values from 0 to 255, i.e. the "I" index value, represent colors from the FLTK 1.3.x standard colormap and are allocated as needed on screens without TrueColor support.

The Fl_Color enumeration type defines the standard colors and color cube for the first 256 colors.

All of these are named with symbols in FLTK/Enumerations.bi.

Example:
Figure 6



Color values greater than 255 are treated as 24-bit RGB values.
These are mapped to the closest color supported by the screen, either from one of the 256 colors in the FLTK 1.3.x colormap or a direct RGB value on TrueColor screens
.

```
private function fl_rgb_color overload (r as ubyte, g as ubyte, b as ubyte) as
Fl_Color
private function fl_rgb_color overload (g as ubyte) as Fl_Color
```

Generate Fl_Color out of specified 8-bit RGB values or one 8-bit grayscale value.

```
private sub fl_color overload(c as Fl_Color)
private sub fl_color overload(c as long)
```

Sets the color for all subsequent drawing operations. Please use the first form: the second form is only provided for back compatibility.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color.

```
private function fl_color() as Fl_Color
```
Returns the last color that was set using fl_color(). This can be used for state save/restore.

```
private sub fl_color overload(r as ubyte, g as ubyte, b as ubyte)
```
Set the color for all subsequent drawing operations. The closest possible match to the RGB color is used.

The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used.

```
declare static function get_color(i as Fl_Color) as unsigned long
declare static sub get_color(i as Fl_Color, byref red as ubyte, byref green as
ubyte, byref blue as ubyte)
```
Generate RGB values from a colormap index value i. The first returns the RGB as a 32-bit unsigned integer, and the second decomposes the RGB into three 8-bit values.

Fl.get_system_colors()
Fl.foreground()
Fl.background()
Fl.background2()

The first gets color values from the user preferences or the system, and the other routines are used to apply those values.

```
declare static sub own_colormap()
declare static sub free_color(i as Fl_Color, overlay as long = 0)
declare static sub set_color(as Fl_Color, as ubyte, as ubyte, as ubyte)
declare static sub set_color(i as Fl_Color, c as unsigned long)
```

`own_colormap()` is used to install a local colormap [X11 only].
`free_color()` and `set_color()` are used to remove and replace entries from the colormap.

There are two predefined graphical interfaces for choosing colors.
The function `fl_show_colormap()` shows a table of colors and returns an Fl_Color index value.

The `Fl_Color_Chooser` widget provides a standard RGB color chooser
.
As the Fl_Color encoding maps to a 32-bit unsigned integer representing RGBI, it is also possible to specify a color using a hex constant as a color map index:

```
// COLOR MAP INDEX
color(&h000000II)
            ------ |
              |    | Color map index (8 bits)
              |
              | Must be zero

button->color(&h000000ff);
// colormap index #255 (FL_WHITE)
```

or specify a color using a hex constant for the RGB components:

```
// RGB COLOR ASSIGNMENTS
color(&hRRGGBB00)
            |    |   |   | Must be zero
            |    |   | Blue (8 bits)
            |    | Green (8 bits)
            | Red (8 bits)

button->color(&hff000000);
button->color(&h00ff0000);
button->color(&h0000ff00);
button->color(&hffffff00);
// RGB: red
// RGB: green
// RGB: blue
// RGB: white
```

Note
If TrueColor is not available, any RGB colors will be set to the nearest entry in the colormap.

Line Dashes and Thickness

FLTK supports drawing of lines with different styles and widths. Full functionality is not available under Windows 95, 98, and Me due to the reduced drawing functionality these operating systems provide.

`private sub fl_line_style(style as long, width_ as long=0, dashes as zstring ptr=0)`
Set how to draw lines (the "pen"). If you change this it is your responsibility to set it back to the default with `fl_line_style(0)`.

Note: Because of how line styles are implemented on MS Windows systems, you must set the line style after setting the drawing color. If you set the color after the line style you will lose the line style settings!

`style` is a bitmask which is a bitwise-OR of the following values. If you don't specify a dash type you will get a solid line.
If you don't specify a cap or join type you will get a system-defined default of whatever value is fastest.

- FL_SOLID ------
- FL_DASH - - - -
- FL_DOT ....
- FL_DASHDOT - .- .
- FL_DASHDOTDOT - ..-
- FL_CAP_FLAT
- FL_CAP_ROUND
- FL_CAP_SQUARE (extends past end point 1/2 line width)
- FL_JOIN_MITER (pointed)
- FL_JOIN_ROUND
- FL_JOIN_BEVEL (flat)

`width` is the number of pixels thick to draw the lines. Zero results in the system-defined default, which on both X and Windows is somewhat different and nicer than 1.

`dashes` is a pointer to an array of dash lengths, measured in pixels.
The first location is how long to draw a solid portion, the next is how long to draw the gap, then the solid, etc. It is terminated with a zero-length entry.

A NULL pointer or a zero-length array results in a solid line.
Odd array sizes are not supported and result in undefined behavior.

Note: The dashes array does not work under Windows 95, 98, or Me, since those operating systems do not support complex line styles.

Drawing Fast Shapes

These functions are used to draw almost all the FLTK widgets. They draw on exact pixel boundaries and are as fast as possible. Their behavior is duplicated exactly on all platforms FLTK is ported. It is undefined whether these are affected by the transformation matrix, so you should only call these while the matrix is set to the identity matrix (the default).

private sub fl_point(x as long, y as long)
Draw a single pixel at the given coordinates.

```
private sub fl_rect overload(x as long, y as long, w as long, h as long)
private sub fl_rect overload(x as long, y as long, w as long, h as long, c as
Fl_Color)
```
Color a rectangle that exactly fills the given bounding box.

```
private sub fl_rectf overload(x as long, y as long, w as long, h as long)
private sub fl_rectf overload(x as long, y as long, w as long, h as long, c as
Fl_Color)
declare sub fl_rectf overload(x as long, y as long, w as long, h as long, r as
ubyte, g as ubyte, b as ubyte)
```
Color a rectangle with "exactly" the passed r,g,b color.

On screens with less than 24 bits of color this is done by drawing a solid-colored block using `fl_draw_image()` so that the correct color shade is produced.

```
private sub fl_rect overload(x as long, y as long, w as long, h as long)
private sub fl_rect overload(x as long, y as long, w as long, h as long, c as
Fl_Color)
```

Draw a 1-pixel border inside this bounding box.

```
private sub fl_line_style(style as long, width_ as long=0, dashes as zstring ptr=0)
private sub fl_line overload(x as long, y as long, x1 as long, y1 as long)
```

Draw one or two lines between the given points.

```
private sub fl_loop overload(x as long, y as long, x1 as long, y1 as long, x2 as
long, y2 as long)
private sub fl_loop overload(x as long, y as long, x1 as long, y1 as long, x2 as
long, y2 as long, x3 as long, y3 as long)
```

Outline a 3 or 4-sided polygon with lines.

```
private sub fl_polygon overload(x as long, y as long, x1 as long, y1 as long, x2 as
long, y2 as long)
private sub fl_polygon overload(x as long, y as long, x1 as long, y1 as long, x2 as
long, y2 as long, x3 as long, y3 as long)
```

Fill a 3 or 4-sided polygon. The polygon must be convex.

```
private sub fl_xyline overload(x as long, y as long, x1 as long)
private sub fl_xyline overload(x as long, y as long, x1 as long, y2 as long)
private sub fl_xyline overload(x as long, y as long, x1 as long, y2 as long, x3 as
long)
```

Draw horizontal and vertical lines. A horizontal line is drawn first, then a vertical, then a horizontal.

```
private sub fl_yxline overload(x as long, y as long, y1 as long)
private sub fl_yxline overload(x as long, y as long, y1 as long, x2 as long)
private sub fl_yxline overload(x as long, y as long, y1 as long, x2 as long, y3 as
long)
```

Draw vertical and horizontal lines. A vertical line is drawn first, then a horizontal, then a vertical.

```
private sub fl_arc overload(x as long, y as long, w as long, h as long, a1 as
double, a2 as double)
private sub fl_arc overload(x as double, y as double, r as double, start as double,
end_ as double)
private sub fl_pie (x as long, y as long, w as long, h as long, a1 as double, a2 as
double)
```

Draw ellipse sections using integer coordinates. These functions match the rather limited circle drawing code provided by X and MS Windows.
The advantage over using `fl_arc()` with floating point coordinates is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hardcoded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box.
The two angles are measured in degrees counter-clockwise from 3'oclock and are the starting and ending angle of the arc, a2 must be greater or equal to a1.

`fl_arc()` draws a series of lines to approximate the arc.
Notice that the integer version of `fl_arc()` has a different number of arguments to the other `fl_arc()` function described later in this chapter.

`fl_pie()` draws a filled-in pie slice.
This slice may extend outside the line drawn by `fl_arc()` to avoid this use w-1 and h-1.

```
declare sub fl_scroll_ alias "fl_scroll"(X as long, Y as long, W as long, H as
long, dx as long, dy as long, draw_area as sub(as any ptr, as long, as long, as
long, as long), data_ as any ptr)
```
Scroll a rectangle and draw the newly exposed portions.
The contents of the rectangular area is first shifted by dx and dy pixels.
The callback is then called for every newly exposed rectangular area.

Drawing Complex Shapes

The complex drawing functions let you draw arbitrary shapes with 2-D linear transformations.
The functionality matches that found in the Adobe® PostScriptTM language.
The exact pixels that are filled are less defined than for the fast drawing functions so that FLTK can take advantage of drawing hardware.

On both X and MS Windows the transformed vertices are rounded to integers before drawing the line segments: this severely limits the accuracy of these functions for complex graphics, so use OpenGL when greater accuracy and/or performance is required.

```
private sub fl_push_matrix()
private sub fl_pop_matrix()
```
Save and restore the current transformation. The maximum depth of the stack is 32 entries.

```
private sub fl_scale overload (x as double, y as double)
private sub fl_scale overload (x as double)
private sub fl_translate (x as double, y as double)
private sub fl_rotate (d as double)
private sub fl_mult_matrix (a as double, b as double, c as double, d as double, x
as double, y as double)
```
Concatenate another transformation onto the current one. The rotation angle is in degrees (not radians) and is counter-clockwise.

```
private function fl_transform_x (x as double, y as double) as double
private function fl_transform_y (x as double, y as double) as double
private function fl_transform_dx (x as double, y as double) as double
private function fl_transform_dy (x as double, y as double) as double
private sub fl_transformed_vertex (xf as double, yf as double)
```
Transform a coordinate or a distance using the current transformation matrix.
After transforming a coordinate pair, it can be added to the vertex list without any further translations using fl_transformed_vertex().

```
private sub fl_begin_points()
private sub fl_end_points()
```
Start and end drawing a list of points. Points are added to the list with fl_vertex().

```
private sub fl_begin_line()
private sub fl_end_line()
```
Start and end drawing lines.

```
private sub fl_begin_loop()
private sub fl_end_loop()
```
Start and end drawing a closed sequence of lines.

```
private sub fl_begin_polygon()
private sub fl_end_polygon()
```
Start and end drawing a convex filled polygon.

```
private sub fl_begin_complex_polygon()
private sub fl_gap()
private sub fl_end_complex_polygon()
```
Start and end drawing a complex filled polygon. This polygon may be concave, may have holes in it, or may be several disconnected pieces. Call `fl_gap()` to separate loops of the path. It is unnecessary but harmless to call `fl_gap()` before the first vertex, after the last one, or several times in a row.

`fl_gap()` should only be called between `fl_begin_complex_polygon()` and `fl_end_complex_polygon()`. To outline the polygon, use `fl_begin_loop()` and replace each `fl_gap()` with a `fl_end_loop() fl_begin_loop()` pair.

Note: For portability, you should only draw polygons that appear the same whether "even/odd" or "nonzero" winding rules are used to fill them. Holes should be drawn in the opposite direction of the outside loop.

```
private sub fl_vertex (x as double, y as double)
```
Add a single vertex to the current path.

```
private sub fl_curve (X0 as double, Y0 as double, X1 as double, Y1 as double, X2 as double, Y2 as double, X3 as double, Y3 as double)
```
Add a series of points on a Bezier curve to the path. The curve ends (and two of the points are) at X0,Y0 and X3,Y3.

```
private sub fl_arc overload(x as double, y as double, r as double, start as double, end_ as double)
```
Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using scale and rotate before calling `fl_arc()`.
The center of the circle is given by x and y, and r is its radius.
`fl_arc()` takes start and end angles that are measured in degrees counter-clockwise from 3 o'clock.
If end is less than start then it draws the arc in a clockwise direction.

```
private sub fl_circle (x as double, y as double, r as double)
```
`fl_circle(...)` is equivalent to `fl_arc(...,0,360)` but may be faster. It must be the only thing in the path: if you want a circle as part of a complex polygon you must use `fl_arc()`.

Note: `fl_circle()` draws incorrectly if the transformation is both rotated and non-square scaled.

Drawing Text

All text is drawn in the current font. It is undefined whether this location or the characters are modified by the current transformation.

```
declare sub fl_draw overload(str_ as const zstring ptr, x as long, y as long)
private sub fl_draw overload(str_ as const zstring ptr, n as long, x as long, y as long)
```

Draw a nul-terminated string or an array of n characters starting at the given location.
Text is aligned to the left and to the baseline of the font.
To align to the bottom, subtract fl_descent() from y.
To align to the top, subtract fl_descent() and add fl_height().
This version of fl_draw() provides direct access to the text drawing function of the underlying OS.
It does not apply any special handling to control characters.

```
declare sub fl_draw overload(str_ as const zstring ptr, x as long, y as long, w as long, h as long, align as Fl_Align, img as Fl_Image ptr=0, draw_symbols as long = 1)
```

Fancy string drawing function which is used to draw all the labels.
The string is formatted and aligned inside the passed box. Handles '\t' and '\n', expands all other control characters to ∧ X, and aligns inside or against the edges of the box described by x, y, w and h.
See Fl_Widget.align() for values for align. The value FL_ALIGN_INSIDE is ignored, as this function always prints inside the box.

If `img` is provided and is not `NULL`, the image is drawn above or below the text as specified by the align value.

The `draw_symbols` argument specifies whether or not to look for symbol names starting with the "@" character.

```
declare sub fl_measure overload(str_ as const zstring ptr, byval x as long, byval y as long, draw_symbols as long = 1)
```

Measure how wide and tall the string will be when printed by the `fl_draw(...align)` function.
This includes leading/trailing white space in the string, kerning, etc.
If the incoming x is non-zero it will wrap to that width.
This will probably give unexpected values unless you have called `fl_font()` explicitly in your own code. Refer to the full documentation for `fl_measure()` for details on usage and how to avoid common pitfalls.

See also
`fl_text_extents()` – measure the 'inked' area of a string
`fl_width()` – measure the pixel width of a string or single character
`fl_height()` – measure the pixel height of the current font
`fl_descent()` – the height of the descender for the current font

```
private function fl_height overload () as long
```

Recommended minimum line spacing for the current font.
You can also just use the value of size passed to `fl_font()`.

See also
`fl_text_extents()`, `fl_measure()`, `fl_width()`, `fl_descent()`

```
private function fl_descent () as long
```
Recommended distance above the bottom of a `fl_height()` tall box to draw the text at so it looks centered vertically in that box.

```
declare function fl_width overload (txt as const zstring ptr) as double
private function fl_width overload (txt as const zstring ptr, n as long) as double
private function fl_width overload (c as unsigned long) as double
```
Return the pixel width of a nul-terminated string, a sequence of n characters, or a single character in the current font.

See also
`fl_measure()`, `fl_text_extents()`, `fl_height()`, `fl_descent()`

```
declare sub fl_text_extents overload (as const zstring ptr, byref dx as long, byref
dy as long, byref w as long, byref h as long)
```
Determines the minimum pixel dimensions of a nul-terminated string, ie. the 'inked area'.
Given a string "txt" drawn using `fl_draw(txt, x, y)` you would determine its pixel extents on the display using `fl_text_extents(txt, dx, dy, wo, ho)` such that a bounding box that exactly fits around the inked area of the text could be drawn with `fl_rect(x+dx, y+dy, wo, ho)`.
Refer to the full documentation for `fl_text_extents()` for details on usage.

See also
`fl_measure()`, `fl_width()`, `fl_height()`, `fl_descent()`

```
declare function  fl_shortcut_label overload(shortcut as unsigned long) as const
zstring ptr
```
Unparse a shortcut value as used by Fl_Button or Fl_Menu_Item into a human-readable string like "Alt+N".

This only works if the shortcut is a character key or a numbered function key.
If the shortcut is zero an empty string is returned.
The return value points at a static buffer that is overwritten with each call.

Fonts

FLTK supports a set of standard fonts based on the Times, Helvetica/Arial, Courier, and Symbol typefaces, as well as custom fonts that your application may load. Each font is accessed by an index into a font table.

Initially only the first 16 faces are filled in.
There are symbolic names for them: FL_HELVETICA, FL_TIMES, FL_COURIER, and modifier values FL_BOLD and FL_ITALIC which can be added to these, and FL_SYMBOL and FL_ZAPF_DINGBATS.
Faces greater than 255 cannot be used in Fl_Widget labels, since Fl_Widget stores the index as a byte.

One important thing to note about 'current font' is that there are so many paths through the GUI event handling code as widgets are partially or completely hidden, exposed and then re-drawn and therefore you can not guarantee that 'current font' contains the same value that you set on the other side of the event loop.
Your value may have been superseded when a widget was redrawn.
You are strongly advised to set the font explicitly before you draw any text or query the width and height of text strings, etc.

```
private sub fl_font overload(face as Fl_Font, fsize as Fl_Fontsize)
```
Set the current font, which is then used by the routines described above.
You may call this outside a draw context if necessary to call `fl_width()`, but on X this will open the display.

The font is identified by a face and a size. The size of the font is measured in pixels and not "points". Lines should be spaced size pixels apart or more.

```
private function fl_font overload() as Fl_Font
private function fl_size () as Fl_Fontsize
```
Returns the face and size set by the most recent call to fl_font(a,b). This can be used to save/restore the font.

Character Encoding

FLTK 1.3 expects all text in Unicode UTF-8 encoding.
UTF-8 is ASCII compatible for the first 128 characters.

International characters are encoded in multibyte sequences.

FLTK expects individual characters, characters that are not part of a string, in UCS-4 encoding, which is also ASCII compatible, but requires 4 bytes to store a Unicode character.

For more information about character encodings, see the chapter on Unicode and UTF-8 Support.

Drawing Overlays

These functions allow you to draw interactive selection rectangles without using the overlay hardware. FLTK will XOR a single rectangle outline over a window.

```
declare sub fl_overlay_rect(x as long, y as long, w as long, h as long)
declare sub fl_overlay_clear()
```
`fl_overlay_rect()` draws a selection rectangle, erasing any previous rectangle by XOR'ing it first. `fl_overlay_clear()` will erase the rectangle without drawing a new one.

Using these functions is tricky. You should make a widget with both a `handle()` and `draw()` method. `draw()` should call `fl_overlay_clear()` before doing anything else. Your `handle()` method should call `window()->make_current()` and then `fl_overlay_rect()` after `FL_DRAG` events, and should call `fl_overlay_clear()` after a `FL_RELEASE` event.

Drawing Images

To draw images, you can either do it directly from data in your memory, or you can create a Fl_Image object. The advantage of drawing directly is that it is more intuitive, and it is faster if the image data changes more often than it is redrawn. The advantage of using the object is that FLTK will cache translated forms of the image (on X it uses a server pixmap) and thus redrawing is much faster.

Direct Image Drawing

The behavior when drawing images when the current transformation matrix is not the identity is not defined, so you should only draw images when the matrix is set to the identity.

```
private sub fl_draw_image overload(buf as const ubyte ptr, X as long, Y as long, W
as long, H as long, D as long=3, L as long=0)

private sub fl_draw_image_mono overload(buf as const ubyte ptr, X as long, Y as
long, W as long, H as long, D as long=1, L as long=0)
```

Draw an 8-bit per color RGB or luminance image.
The pointer points at the "r" data of the top-left pixel.
Color data must be in r,g,b order.
The top left corner is given by X and Y and the size of the image is given by W and H.
D is the delta to add to the pointer between pixels, it may be any value greater or equal to 3, or it can be negative to flip the image horizontally.
L is the delta to add to the pointer between lines (if 0 is passed it uses W*D). and may be larger than W*D to crop data, or negative to flip the image vertically.

It is highly recommended that you put the following code before the first `show()` of any window in your program to get rid of the dithering if possible:

Fl.visual(FL_RGB);

Gray scale (1-channel) images may be drawn.
This is done if abs(D) is less than 3, or by calling `fl_draw_image_mono()`.
Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used.
Setting D greater than 1 will let you display one channel of a color image.

Note: The X version does not support all possible visuals.
If FLTK cannot draw the image in the current visual it will abort.
FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

```
private sub fl_draw_image overload(cb as Fl_Draw_Image_Cb, data_ as any ptr, X as
long, Y as long, W as long, H as long, D as long=3)

private sub fl_draw_image_mono overload(cb as Fl_Draw_Image_Cb, data_ as any ptr, X
as long, Y as long, W as long, H as long, D as long=1)
```

Call the passed function to provide each scan line of the image. This lets you generate the image as it is being drawn, or do arbitrary decompression of stored data, provided it can be decompressed to individual scan lines easily.

The callback is called with the user data pointer which can be used to point at a structure of information about the image, and the x, y, and w of the scan line desired from the image. 0,0 is the upper-left corner of the image, not X,Y.
A pointer to a buffer to put the data into is passed. You must copy w pixels from scanline y, starting at pixel x, to this buffer.
Due to cropping, less than the whole image may be requested. So x may be greater than zero, the first y may be greater than zero, and w may be less than W. The buffer is long enough to store the entire W*D pixels, this is for convenience with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if x is not zero, copy the data over so the x'th pixel is at the start of the buffer.

You can assume the y's will be consecutive, except the first one may be greater than zero.
If D is 4 or more, you must fill in the unused bytes with zero.

```
declare function fl_draw_pixmap overload(data_ as zstring const ptr const ptr, x as
long , y as long, as Fl_Color=FL_GRAY) as long

declare function fl_draw_pixmap overload(cdata as const zstring const ptr const
ptr, x as long , y as long, as Fl_Color=FL_GRAY) as long
```

Draws XPM image data, with the top-left corner at the given position. The image is dithered on 8-bit displays so you won't lose color space for programs displaying both images and pixmaps. This function returns zero if there was any error decoding the XPM data.

Transparent colors are replaced by the optional Fl_Color argument.
To draw with true transparency you must use the Fl_Pixmap class.

```
declare function fl_measure_pixmap overload(data_ as zstring const ptr const ptr,
byref w as long , byref h as long) as long

declare function fl_measure_pixmap overload(cdata as const zstring const ptr const
ptr, byref w as long , byref h as long) as long
```

An XPM image contains the dimensions in its data. This function finds and returns the width and height.

The return value is non-zero if the dimensions were parsed ok and zero if there was any problem.

Direct Image Reading

FLTK provides a single function for reading from the current window or off-screen buffer into a RGB(A) image buffer.

```
declare function fl_read_image(p as ubyte ptr, X as long, Y as long, W as long, H
as long, alpha as long=0) as ubyte ptr
```

Read a RGB(A) image from the current window or off-screen buffer. The p argument points to a buffer that can hold the image and must be at least W * H * 3 bytes when reading RGB images and W * H * 4 bytes when reading RGBA images. If NULL, fl_read_image() will create an array of the proper size which can be freed using delete[].

The alpha parameter controls whether an alpha channel is created and the value that is placed in the alpha channel. If 0, no alpha channel is generated.

Image Classes

FLTK provides a base image class called Fl_Image which supports creating, copying, and drawing images of various kinds, along with some basic color operations.
Images can be used as labels for widgets using the image() and deimage() methods or drawn directly.

The Fl_Image class does almost nothing by itself, but is instead supported by three basic image types:
• Fl_Bitmap
• Fl_Pixmap
• Fl_RGB_Image

The Fl_Bitmap class encapsulates a mono-color bitmap image.
The draw() method draws the image using the current drawing color.

The Fl_Pixmap class encapsulates a colormapped image. The draw() method draws the image using the colors in the file, and masks off any transparent colors automatically.

The Fl_RGB_Image class encapsulates a full-color (or grayscale) image with 1 to 4 color components. Images with an even number of components are assumed to contain an alpha channel that is used for transparency.
The transparency provided by the draw() method is either a 24-bit blend against the existing window contents or a "screen door" transparency mask, depending on the platform and screen color depth.

```
declare function fl_can_do_alpha_blending() as byte
```

fl_can_do_alpha_blending() will return 1, if your platform supports true alpha blending for RGBA images, or 0, if FLTK will use screen door transparency.

FLTK also provides several image classes based on the three standard image types for common file formats:

• Fl_GIF_Image
• Fl_JPEG_Image
• Fl_PNG_Image
• Fl_PNM_Image
• Fl_XBM_Image
• Fl_XPM_Image

Each of these image classes loads a named file of the corresponding format. The Fl_Shared_Image class can be used to load any type of image file - the class examines the file and constructs an image of the appropriate type.

It can also be used to scale an image to a certain size in drawing units, independently from its size in pixels (see Fl_Shared_Image.scale()).

Finally, FLTK provides a special image class called Fl_Tiled_Image to tile another image object in the specified area.

This class can be used to tile a background image in a Fl_Group widget, for example.

```
declare virtual function copy(W as long, H as long) as Fl_Image ptr
declare function copy() as Fl_Image ptr
```

The copy() method creates a copy of the image. The second form specifies the new size of the image - the image is resized using the nearest-neighbor algorithm (this is the default).

Note
As of FLTK 1.3.3 the image resizing algorithm can be changed.

See Fl_Image.RGB_scaling(Fl_RGB_Scaling method)

```
declare virtual sub draw(X as long, Y as long, W as long, H as long, cx as long=0,
cy as long=0)
```

The draw() method draws the image object. x,y,w,h indicates the destination rectangle. cx,cy,w,h is the source rectangle.
This source rectangle is copied to the destination.
The source rectangle may extend outside the image, i.e. ox and oy may be negative and w and h may be bigger than the image, and this area is left unchanged.

Note
See exceptions for Fl_Tiled_Image.draw() regarding arguments cx, cy, w, and h.

```
declare sub draw(X as long, Y as long)
```

Draws the image with the upper-left corner at x, y.
This is the same as doing `img->draw(x, y, img->w(), img->h(), 0, 0)` where `img` is a pointer to any Fl_Image type.

Handling Events

This chapter discusses the FLTK event model and how to handle events in your program or widget.

The FLTK Event Model

Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application.

Events can also come from other programs like the window manager.

Events are identified by the integer argument passed to a handle() method that overrides the Fl_Widget.handle() virtual method.

Other information about the most recent event is stored in static locations and acquired by calling the Fl.event_*() methods.

This static information remains valid until the next event is read from the window system, so it is ok to look at it outside of the handle() method.

Event numbers can be converted to their actual names using the fl_eventnames[] array defined in FLTK/names.bi see next chapter for details.

In the next chapter, the MyClass.handle() example shows how to override the Fl_Widget.handle() method to accept and process specific events.

Mouse Events

FL_PUSH

A mouse button has gone down with the mouse pointing at this widget.
You can find out what button by calling Fl.event_button().
You find out the mouse position by calling Fl.event_x() and Fl.event_y().

A widget indicates that it "wants" the mouse click by returning non-zero from its handle() method, as in the MyClass.handle() example. It will then become the Fl.pushed() widget and will get FL_DRAG and the matching FL_RELEASE events.
If handle() returns zero then FLTK will try sending the FL_PUSH to another widget.

FL_DRAG

The mouse has moved with a button held down. The current button state is in Fl.event_state().
The mouse position is in Fl.event_x() and Fl.event_y().

In order to receive FL_DRAG events, the widget must return non-zero when handling FL_PUSH.

FL_RELEASE

A mouse button has been released. You can find out what button by calling Fl.event_button().
In order to receive the FL_RELEASE event, the widget must return non-zero when handling FL_PUSH.

FL_MOVE

The mouse has moved without any mouse buttons held down. This event is sent to the Fl.belowmouse() widget.
In order to receive FL_MOVE events, the widget must return non-zero when handling FL_ENTER.

FL_MOUSEWHEEL

The user has moved the mouse wheel. The Fl.event_dx() and Fl.event_dy() methods can be used to find the amount to scroll horizontally and vertically.

Focus Events

FL_ENTER

The mouse has been moved to point at this widget. This can be used for highlighting feedback. If a widget wants to highlight or otherwise track the mouse, it indicates this by returning non-zero from its handle() method.
It then becomes the Fl.belowmouse() widget and will receive FL_MOVE and FL_LEAVE events.

FL_LEAVE

The mouse has moved out of the widget.
In order to receive the FL_LEAVE event, the widget must return non-zero when handling FL_ENTER.

FL_FOCUS

This indicates an attempt to give a widget the keyboard focus.
If a widget wants the focus, it should change itself to display the fact that it has the focus, and return non-zero from its handle() method. It then becomes the Fl.focus() widget and gets FL_KEYDOWN, FL_KEYUP, and FL_UNFOCUS events.

The focus will change either because the window manager changed which window gets the focus, or because the user tried to navigate using tab, arrows, or other keys. You can check Fl.event_key() to figure out why it moved. For navigation it will be the key pressed and for interaction with the window manager it will be zero.

FL_UNFOCUS

This event is sent to the previous Fl.focus() widget when another widget gets the focus or the window loses focus.

Keyboard Events

FL_KEYBOARD, FL_KEYDOWN, FL_KEYUP

A key was pressed (FL_KEYDOWN) or released (FL_KEYUP). FL_KEYBOARD is a synonym for FL_KEYDOWN, and both names are used interchangeably in this documentation.

The key can be found in Fl.event_key(). The text that the key should insert can be found with Fl.event_text() and its length is in Fl.event_length().

If you use the key, then handle() should return 1.
If you return zero then FLTK assumes you ignored the key and will then attempt to send it to a parent widget.
If none of them want it, it will change the event into a FL_SHORTCUT event.

FL_KEYBOARD events are also generated by the character palette/map.
To receive FL_KEYBOARD events you must also respond to the FL_FOCUS and FL_UNFOCUS events by returning 1. This way FLTK knows whether to bother sending your widget keyboard events.

(Some widgets don't need them, e.g. Fl_Box.)

If you are writing a text-editing widget you may also want to call the Fl.compose() function to translate individual keystrokes into characters.

FL_KEYUP events are sent to the widget that currently has focus.
This is not necessarily the same widget that received the corresponding FL_KEYDOWN event because focus may have changed between events.

Todo Add details on how to detect repeating keys, since on some X servers a repeating key will generate both FL_KEYUP and FL_KEYDOWN, such that to tell if a key is held, you need Fl.event_key(long) to detect if the key is being held down during FL_KEYUP or not.

FL_SHORTCUT

If the Fl.focus() widget is zero or ignores an FL_KEYBOARD event then FLTK tries sending this event to every widget it can, until one of them returns non-zero.
FL_SHORTCUT is first sent to the Fl.belowmouse() widget, then its parents and siblings, and eventually to every widget in the window, trying to find an object that returns non-zero.

FLTK tries really hard to not to ignore any keystrokes!
You can also make "global" shortcuts by using Fl.add_handler().
A global shortcut will work no matter what windows are displayed or which one has the focus.

Widget Events

FL_DEACTIVATE

This widget is no longer active, due to deactivate() being called on it or one of its parents. Please note that although active() may still return true for this widget after receiving this event, it is only truly active if active() is true for both it and all of its parents. (You can use active_r() to check this).

FL_ACTIVATE

This widget is now active, due to activate() being called on it or one of its parents.

FL_HIDE

This widget is no longer visible, due to hide() being called on it or one of its parents, or due to a parent window being minimized. Please note that although visible() may still return true for this widget after receiving this event, it is only truly visible if visible() is true for both it and all of its parents. (You can use visible_r() to check this).

FL_SHOW

This widget is visible again, due to show() being called on it or one of its parents, or due to a parent window being restored. A child Fl_Window will respond to this by actually creating the window if not done already, so if you subclass a window, be sure to pass FL_SHOW to the base class handle() method!

Note
The events in this chapter ("Widget Events"), i.e. FL_ACTIVATE, FL_DEACTIVATE, FL_SHOW, and FL_HIDE, are the only events deactivated and invisible widgets can usually get, depending on their states.
Under certain circumstances, there may also be FL_LEAVE or FL_UNFOCUS events delivered to deactivated or hidden widgets.

Clipboard Events

FL_PASTE

You should get this event some time after you call Fl.paste(). The contents of Fl.event_text() is the text to insert and the number of characters is in Fl.event_length().

FL_SELECTIONCLEAR

The Fl.selection_owner() will get this event before the selection is moved to another widget. This indicates that some other widget or program has claimed the selection. Motif programs used this to clear the selection indication. Most modern programs ignore this.

Drag and Drop Events

FLTK supports drag and drop of text and files from any application on the desktop to an FLTK widget. Text is transferred using UTF-8 encoding. Files are received as a list of full path and file names, separated by newline.

On some X11 platforms, files are received as a URL-encoded UTF-8 string, that is, non-ASCII bytes (and a few others such as space and %) are replaced by the 3 bytes "%XY" where XY are the byte's hexadecimal value. The fl_decode_uri() function can be used to transform in-place the received string into a proper UTF-8 string. On these platforms, strings corresponding to dropped files are further prepended by file:// (or other prefixes such as computer://).

See Fl.dnd() for drag and drop from an FLTK widget.
The drag and drop data is available in Fl.event_text() at the concluding FL_PASTE.
On some platforms, the event text is also available for the FL_DND_* events, however application must not depend on that behavior because it depends on the protocol used on each platform.

FL_DND_* events cannot be used in widgets derived from Fl_Group or Fl_Window.

FL_DND_ENTER

The mouse has been moved to point at this widget. A widget that is interested in receiving drag'n'drop data must return 1 to receive FL_DND_DRAG, FL_DND_LEAVE and FL_DND_RELEASE events.

FL_DND_DRAG

The mouse has been moved inside a widget while dragging data. A widget that is interested in receiving drag'n'drop data should indicate the possible drop position.

FL_DND_LEAVE

The mouse has moved out of the widget.

FL_DND_RELEASE

The user has released the mouse button dropping data into the widget. If the widget returns 1, it will receive the data in the immediately following FL_PASTE event.

Other events

FL_SCREEN_CONFIGURATION_CHANGED

Sent whenever the screen configuration changes (a screen is added/removed, a screen resolution is changed, screens are moved). Use Fl.add_handler() to be notified of this event.

FL_FULLSCREEN

The application window has been changed from normal to fullscreen, or from fullscreen to normal. If you are using a X window manager which supports Extended Window Manager Hints, this event will not be delivered until the change has actually happened.

Fl.event_*() methods

FLTK keeps the information about the most recent event in static storage. This information is good until the next event is processed. Thus it is valid inside handle() and callback() methods.

These are all trivial inline functions and thus very fast and small:
- Fl.event_button()
- Fl.event_clicks()
- Fl.event_dx()
- Fl.event_dy()
- Fl.event_inside()
- Fl.event_is_click()
- Fl.event_key()
- Fl.event_length()
- Fl.event_state()
- Fl.event_text()
- Fl.event_x()
- Fl.event_x_root()
- Fl.event_y()
- Fl.event_y_root()
- Fl.get_key()
- Fl.get_mouse()
- Fl.test_shortcut()

Event Propagation

Widgets receive events via the virtual handle() function. The argument indicates the type of event that can be handled. The widget must indicate if it handled the event by returning 1. FLTK will then remove the event and wait for further events from the host. If the widget's handle function returns 0, FLTK may redistribute the event based on a few rules.

Most events are sent directly to the handle() method of the Fl_Window that the window system says they belong to.
The window (actually the Fl_Group that Fl_Window is a subclass of) is responsible for sending the events on to any child widgets. To make the Fl_Group code somewhat easier, FLTK sends some events (FL_DRAG, Fl_RELEASE, FL_KEYBOARD, FL_SHORTCUT, FL_UNFOCUS, and FL_LEAVE) directly to leaf widgets.
These procedures control those leaf widgets:
- Fl.add_handler()
- Fl.belowmouse()
- Fl.focus()
- Fl.grab()
- Fl.modal()
- Fl.pushed()
- Fl.release() (deprecated, see Fl.grab(0))
- Fl_Widget.take_focus()

FLTK propagates events along the widget hierarchy depending on the kind of event and the status of the UI. Some events are injected directly into the widgets, others may be resent as new events to a different group of receivers.

Mouse click events are first sent to the window that caused them. The window then forwards the event down the hierarchy until it reaches the widget that is below the click position. If that widget uses the given event, the widget is marked "pushed" and will receive all following mouse motion (FL_DRAG) events until the mouse button is released.

Mouse motion (FL_MOVE) events are sent to the Fl.belowmouse() widget, i.e. the widget that returned 1 on the last FL_ENTER event.
Mouse wheel events are sent to the window that caused the event. The window propagates the event down the tree, first to the widget that is below the mouse pointer, and if that does not succeed, to all other widgets in the group. This ensures that scroll widgets work as expected with the widget furthest down in the hierarchy getting the first opportunity to use the wheel event, but also giving scroll bars, that are not directly below the mouse a chance.

Keyboard events are sent directly to the widget that has keyboard focus. If the focused widget rejects the event, it is resent as a shortcut event, first to the top-most window, then to the widget below the mouse pointer, propagating up the hierarchy to all its parents. Those send the event also to all widgets that are not below the mouse pointer.
Now if that did not work out, the shortcut is sent to all registered shortcut handlers.
If we are still unsuccessful, the event handler flips the case of the shortcut letter and starts over. Finally, if the key is "escape", FLTK sends a close event to the top-most window.

All other events are pretty much sent right away to the window that created the event.
Widgets can "grab" events. The grabbing window gets all events exclusively, but usually by the same rules as described above.
Windows can also request exclusivity in event handling by making the window modal.

FLTK Compose-Character Sequences

The character composition done by Fl_Input widget requires that you call the Fl.compose() function if you are writing your own text editor widget.
Currently, all characters made by single key strokes with or without modifier keys, or by system-defined character compose sequences (that can involve dead keys or a compose key) can be input. You should call Fl.compose() in case any enhancements to this processing are done in the future. The interface has been designed to handle arbitrary UTF-8 encoded text.

The following methods are provided for character composition:
• Fl.compose()
• Fl.compose_reset()

Under Mac OS X, FLTK "previews" partially composed sequences.

Advanced FLTK

This chapter explains advanced programming and design topics that will help you to get the most out of FLTK.

Multithreading

FLTK can be used to implement a GUI for a multithreaded application but, as with multithreaded programming generally, there are some concepts and caveats that must be kept in mind.

Key amongst these is that, for many of the target platforms on which FLTK is supported, only the main() thread of the process is permitted to handle system events, create or destroy windows and open or close windows.
Further, only the main() thread of the process can safely write to the display.

To support this in a portable way, all FLTK draw() methods are executed in the main() thread.
A worker thread may update the state of an existing widget, but it may not do any rendering directly, nor create or destroy a window.

(NOTE: A special case exists for Fl_Gl_Window where it can, with suitable precautions, be possible to safely render to an existing GL context from a worker thread.)

Creating portable threads

We do not provide a threading interface as part of the library.

FLTK has been used with a variety of thread interfaces, so if the simple example shown in test/threads.cxx does not cover your needs, you might want to select a third-party library that provides the features you require.

FLTK multithread locking – Fl.lock() and Fl.unlock()

In a multithreaded program, drawing of widgets (in the main() thread) happens asynchronously to widgets being updated by worker threads, so no drawing can occur safely whilst a widget is being modified (and no widget should be modified whilst drawing is in progress).

FLTK supports multithreaded applications using a locking mechanism internally. This allows a worker thread to lock the rendering context, preventing any drawing from taking place, whilst it changes the value of its widget.

Note
The converse is also true; whilst a worker thread holds the lock, the main() thread may not be able to process any drawing requests, nor service any events. So a worker thread that holds the FLTK lock must contrive to do so for the shortest time possible or it could impair operation of the application.

The lock operates broadly as follows.
Using the FLTK library, the main() thread holds the lock whenever it is processing events or redrawing the display.
It acquires (locks) and releases (unlocks) the FLTK lock automatically and no "user intervention" is required.

Indeed, a function that runs in the context of the main() thread ideally should not acquire / release the FLTK lock explicitly. (Though note that the lock calls are recursive, so calling Fl.lock() from a thread that already holds the lock, including the main() thread, is benign. The only constraint is that every call to Fl.lock() must be balanced by a corresponding call to Fl.unlock() to ensure the lock count is preserved.)

The main() thread must call Fl.lock() once before any windows are shown, to enable the internal lock (it is "off" by default since it is not useful in single-threaded applications) but thereafter the main() thread lock is managed by the library internally.

A worker thread, when it wants to alter the value of a widget, can acquire the lock using Fl.lock(), update the widget, then release the lock using Fl.unlock(). Acquiring the lock ensures that the worker thread can update the widget, without any risk that the main() thread will attempt to redraw the widget whilst it is being updated.

Note that acquiring the lock is a blocking action; the worker thread will stall for as long as it takes to acquire the lock.

If the main() thread is engaged in some complex drawing operation this may block the worker thread for a long time, effectively serializing what ought to be parallel operations. (This frequently comes as a surprise to coders less familiar with multithreaded programming issues; see the discussion of "lockless programming" later for strategies for managing this.)

To incorporate the locking mechanism in the library, FLTK must be compiled with -enable-threads set during the configure process. IDE-based versions of FLTK are automatically compiled with the locking mechanism incorporated if possible. Since version 1.3, the configure script that builds the FLTK library also sets -enable-threads by default.

Simple multithreaded examples using Fl.lock

In main(), call Fl.lock() once before Fl.run() or Fl.wait() to enable the lock and start the runtime multithreading support for your program. All callbacks and derived functions like handle() and draw() will now be properly locked.

This might look something like this:

Project 005-threads01 – threads.bas

```
#include "FLTK/Fl.bi"
#include "FLTK/Fl_Window.bi"

Function main() As Long
      Dim result as long
      Dim As ZString * 13 str1 => "hello, world"
      Dim fWindow as Fl_Window = Fl_Window(0, 0, 340, 180, str1)
      '' add widgets here
      fWindow.end_()

      '' Create your windows and widgets here
      Fl.lock() '' "start" the FLTK lock mechanism

      '' show your window
      fWindow.show()

      '' start your worker threads
            ''  start threads …

      '' Run the FLTK main loop
      result = Fl.run_()

      '' terminate any pending worker threads
            '' stop threads …

      return result
End Function
'' call main() and return the error code to the OS
End main()
```

You can start as many threads as you like. From within a thread (other than the main() thread) FLTK calls must be wrapped with calls to Fl.lock() and Fl.unlock():

```
Sub my_thread()
 while (thread_still_running)
   '' do thread work
   ...
   '' compute new values for widgets */
   ...
   Fl.lock()   '' acquire the lock
   my_widget->update(values)
   Fl.unlock()'' release the lock;
               '' allow other threads to access FLTK again
   Fl.awake()'' use Fl.awake() to signal main thread to refresh the GUI
 Wend
End Sub
```

Note

To trigger a refresh of the GUI from a worker thread, the worker code should call Fl.awake()

Using Fl.awake thread messages

You can send messages from worker threads to the main() thread using Fl.awake(void * message). If using this thread message interface, your main() might look like this:

```
1.  #include "FLTK/Fl.bi"
2.  #include "FLTK/Fl_Window.bi"
3.
4.  Function main() As Long
5.        Dim result as long
6.        Dim next_message as any ptr
7.        Dim As ZString * 13 str1 => "hello, world"
8.        Dim fWindow as Fl_Window = Fl_Window(0, 0, 340, 180, str1)
9.        '' add widgets here
10.       fWindow.end_()
11.
12.       '' Create your windows and widgets here
13.       Fl.lock() '' "start" the FLTK lock mechanism
14.
15.       '' show your window
16.       fWindow.show()
17.
18.       '' start your worker threads
19.             ''  start threads …
20.
21.       '' Run the FLTK main loop
22.       '' Run the FLTK loop and process thread messages */
23.       while (Fl.wait() > 0)
24.         if ((next_message = Fl.thread_message()) <> NULL) Then
25.            '' process your data, update widgets, etc.
26.            ...
27.         End If
28.       Wend
29.       '' terminate any pending worker threads
30.       ... stop threads ...
31.       return 0
32. End Function
33. '' call main() and return the error code to the OS
34. End main()
```

Your worker threads can send messages to the main() thread using Fl.awake(message as any ptr):

```
msg as any ptr
Fl.awake(msg)
'' "msg" is a pointer to your message
'' send "msg" to main thread
```

A message can be anything you like. The main() thread can retrieve the message by calling Fl.thread_message().

Using Fl.awake callback messages
You can also request that the main() thread call a function on behalf of the worker thread by using `Fl.awake(cb as Fl_Awake_Handler, userdata as any ptr)`.

The main() thread will execute the callback "as soon as possible" when next processing the pending events. This can be used by a worker thread to perform operations (for example showing or hiding windows) that are prohibited in a worker thread.

```
Sub do_something_cb(userdata as any ptr)
'' Will run in the context of the main thread
... do_stuff ...
End Sub
'' running in worker thread
data as any ptr
'' "data" is a pointer to your user data
Fl.awake(do_something_cb, data); '' call to execute cb in main thread
```

Note
The main() thread will execute the Fl_Awake_Handler callback do_something_cb asynchronously to the worker thread, at some short but indeterminate time after the worker thread registers the request. When it executes the Fl_Awake_Handler callback, the main() thread will use the contents of `userdata` at the time of execution, not necessarily the contents that `userdata` had at the time that the worker thread posted the callback request.
The worker thread should therefore contrive not to alter the contents of `userdata` once it posts the callback, since the worker thread does not know when the main() thread will consume that data. It is often useful that userdata point to a struct, one member of which the main() thread can modify to indicate that it has consumed the data, thereby allowing the worker thread to re-use or update userdata.

Warning
The mechanisms used to deliver `Fl.awake(message as any ptr)` and `Fl.awake(Fl_Awake_Handler cb, user-data as any ptr)` events to the main() thread can interact in unexpected ways on some platforms.
Therefore, for reliable operation, it is advised that a program use either `Fl.awake(Fl_Awake_Handler cb, userdata as any ptr)` or `Fl.awake(message as any ptr)`, but that they never be intermixed. Calling `Fl.awake()` with no parameters should be safe in either case.

If you have to choose between using the `Fl.awake(message as any ptr)` and `Fl.awake(Fl_Awake_Handler cb, userdata as any ptr)` mechanisms and don't know which to choose, then try the `Fl.awake(Fl_Awake_Handler cb, userdata as any ptr)` method first as it tends to be more powerful in general.

FLTK multithreaded "lockless programming"

The simple multithreaded examples shown above, using the FLTK lock, work well for many cases where multiple threads are required. However, when that model is extended to more complex programs, it often produces results that the developer did not anticipate.

A typical case might go something like this. A developer creates a program to process a huge data set. The program has a main() thread and 7 worker threads and is targeted to run on an 8-core computer. When it runs, the program divides the data between the 7 worker threads, and as they process their share of the data, each thread updates its portion of the GUI with the results, locking and unlocking as they do so.

But when this program runs, it is much slower than expected and the developer finds that only one of the eight CPU cores seems to be utilised, despite there being 8 threads in the program. What happened?

The threads in the program all run as expected, but they end up being serialized (that is, not able to run in parallel) because they all depend on the single FLTK lock. Acquiring (and releasing) that lock has an associated cost, and is a blocking action if the lock is already held by any other worker thread or by the main() thread.

If the worker threads are acquiring the lock "too often", then the lock will always be held somewhere and every attempt by any other thread (even main()) to lock will cause that other thread (including main()) to block. And blocking main() also blocks event handling, display refresh...

As a result, only one thread will be running at any given time, and the multithreaded program is effectively reduced to being a (complicated and somewhat less efficient) single thread program.
A "solution" is for the worker threads to lock "less often", such that they do not block each other or the main() thread. But judging what constitutes locking "too often" for any given configuration, and hence will block, is a very tricky question. What works well on one machine, with a given graphics card and CPU configuration may behave very differently on another target machine.

There are "interesting" variations on this theme, too: for example it is possible that a "faulty" multithreaded program such as described above will work adequately on a single-core machine (where all threads are inherently serialized anyway and so are less likely to block each other) but then stall or even deadlock in unexpected ways on a multicore machine when the threads do interfere with each other. (I have seen this - it really happens.)

The "better" solution is to avoid using the FLTK lock so far as possible. Instead, the code should be designed so that the worker threads do not update the GUI themselves and therefore never need to acquire the FLTK lock.
This would be FLTK multithreaded "lockless programming".
There are a number of ways this can be achieved (or at least approximated) in practice but the most direct approach is for the worker threads to make use of the Fl.awake(Fl_Awake_Handler cb, void * userdata) method so that GUI updates can all run in the context of the main() thread, alleviating the need for the worker thread to ever lock. The onus is then on the worker threads to manage the userdata so that it is delivered safely to the main() thread, but there are many ways that can be done.

Note
Using Fl.awake is not, strictly speaking, entirely "lockless" since the awake handler mechanism incorporates resource locking internally to protect the queue of pending awake messages. These resource locks are held transiently and generally do not trigger the pathological blocking issues described here.

However, aside from using Fl.awake, there are many other ways that a "lockless" design can be implemented, including message passing, various forms of IPC, etc.

If you need high performing multithreaded programming, then take some time to study the options and understand the advantages and disadvantages of each; we can't even begin to scratch the surface of this huge topic here!

And of course occasional, sparse, use of the FLTK lock from worker threads will do no harm; it is "excessive" locking (whatever that might be) that triggers the failing behaviour.

It is always a Good Idea to update the GUI at the lowest rate that is acceptable when processing bulk data (or indeed, in all cases!) Updating at a few frames per second is probably adequate for providing feedback during a long calculation. At the upper limit, anything faster than the frame rate of your monitor and the updates will never even be displayed; why waste CPU computing pixels that you will never show?

FLTK multithreaded Constraints

FLTK supports multiple platforms, some of which allow only the main() thread to handle system events and open or close windows. The safe thing to do is to adhere to the following rules for threads on all operating systems:

• Don't show() or hide() anything that contains Fl_Window based widgets from a worker thread. This includes any windows, dialogs, file choosers, subwindows or widgets using Fl_Gl_Window. Note that this constraint also applies to non-window widgets that have tooltips, since the tooltip will contain a Fl_Window object. The safe and portable approach is never to call show() or hide() on any widget from the context of a worker thread. Instead you can use the Fl_Awake_Handler variant of Fl.awake() to request the main() thread to create, destroy, show or hide the widget on behalf of the worker thread.

• Don't call Fl.run(), Fl.wait(), Fl.flush(), Fl.check() or any related methods that will handle system messages from a worker thread

• Don't intermix use of `Fl.awake(Fl_Awake_Handler cb, userdata as any ptr)` and `Fl.awake(message as any ptr)` calls in the same program as they may interact unpredictably on some platforms; choose one or other style of Fl.awake(<thing>) mechanism and use that. (Intermixing calls to Fl.awake() should be safe with either however.)

• Don't start or cancel timers from a worker thread

• Don't change window decorations or titles from a worker thread

• The make_current() method will probably not work well for regular windows, but should always work for a Fl_Gl_Window to allow for high speed rendering on graphics cards with multiple pipelines.

Managing thread-safe access to the GL pipelines is left as an exercise for the reader! (And may be target specific…)

See also: `Fl.lock(), Fl.unlock(), Fl.awake(), Fl.awake(Fl_Awake_Handler cb, userdata as any ptr), Fl.awake(void*message), Fl.thread_message()`.

Unicode and UTF-8 Support

This chapter explains how FLTK handles international text via Unicode and UTF-8.
Unicode support was only recently added to FLTK and is still incomplete. This chapter is Work in Progress, reflecting the current state of Unicode support.

About Unicode, ISO 10646 and UTF-8

The summary of Unicode, ISO 10646 and UTF-8 given below is deliberately brief and provides just enough information for the rest of this chapter.

For further information, please see:
• http://www.unicode.org
• http://www.iso.org
• http://en.wikipedia.org/wiki/Unicode
• http://www.cl.cam.ac.uk/~mgk25/unicode.html
• http://www.apps.ietf.org/rfc/rfc3629.html

The Unicode Standard
The Unicode Standard was originally developed by a consortium of mainly US computer manufacturers and developers of multi-lingual software. It has now become a defacto standard for character encoding and is supported by most of the major computing companies in the world.
Before Unicode, many different systems, on different platforms, had been developed for encoding characters for different languages, but no single encoding could satisfy all languages. Unicode provides access to over 100,000 characters used in all the major languages written today, and is independent of platform and language.

Unicode also provides higher-level concepts needed for text processing and typographic publishing systems, such as algorithms for sorting and comparing text, composite character and text rendering, right-to-left and bi-directional text handling.

Note
There are currently no plans to add this extra functionality to FLTK.

ISO 10646
The International Organisation for Standardization (ISO) had also been trying to develop a single unified character set. Although both ISO and the Unicode Consortium continue to publish their own standards, they have agreed to coordinate their work so that specific versions of the Unicode and ISO 10646 standards are compatible with each other.
The international standard ISO 10646 defines the Universal Character Set (UCS) which contains the characters required for almost all known languages. The standard also defines three different implementation levels specifying how these characters can be combined.

Note
There are currently no plans for handling the different implementation levels or the combining characters in FLTK.

In UCS, characters have a unique numerical code and an official name, and are usually shown using 'U+' and the code in hexadecimal, e.g. U+0041 is the "Latin capital letter A". The UCS characters

U+0000 to U+007F correspond to US-ASCII, and U+0000 to U+00FF correspond to ISO 8859-1 (Latin1).

ISO 10646 was originally designed to handle a 31-bit character set from U+00000000 to U+7FFFFFFF, but the current idea is that 21 bits will be sufficient for all future needs, giving characters up to U+10FFFF. The complete character set is sub-divided into planes. Plane 0, also known as the Basic Multilingual Plane (BMP), ranges from U+0000 to U+FFFD and consists of the most commonly used characters from previous encoding standards.

Other planes contain characters for specialist applications.

The UCS also defines various methods of encoding characters as a sequence of bytes. UCS-2 encodes Unicode characters into two bytes, which is wasteful if you are only dealing with ASCII or Latin1 text, and insufficient if you need characters above U+00FFFF. UCS-4 uses four bytes, which lets it handle higher characters, but this is even more wasteful for ASCII or Latin1.

UTF-8

The Unicode standard defines various UCS Transformation Formats (UTF). UTF-16 and UTF-32 are based on units of two and four bytes. UCS characters requiring more than 16 bits are encoded using "surrogate pairs" in UTF-16.

UTF-8 encodes all Unicode characters into variable length sequences of bytes. Unicode characters in the 7-bit ASCII range map to the same value and are represented as a single byte, making the transformation to Unicode quick and easy.

All UCS characters above U+007F are encoded as a sequence of several bytes. The top bits of the first byte are set to show the length of the byte sequence, and subseqent bytes are always in the range 0x80 to 0xBF.

This combination provides some level of synchronisation and error detection.

Unicode range	Byte sequences
U+00000000 - U+0000007F	0xxxxxxx
U+00000080 - U+000007FF	110xxxxx 10xxxxxx
U+00000800 - U+0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U+00010000 - U+001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U+00200000 - U+03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U+04000000 - U+7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Moving from ASCII encoding to Unicode will allow all new FLTK applications to be easily internationalized and used all over the world. By choosing UTF-8 encoding, FLTK remains largely source-code compatible to previous iterations of the library.

Unicode in FLTK

FLTK will be entirely converted to Unicode using UTF-8 encoding. If a different encoding is required by the underlying operating system, FLTK will convert the string as needed.

It is important to note that the initial implementation of Unicode and UTF-8 in FLTK involves three important areas:
• provision of Unicode character tables and some simple related functions;
• conversion of char * variables and function parameters from single byte per character representation to UTF-8 variable length sequences;
• modifications to the display font interface to accept general Unicode character or UCS code numbers instead of just ASCII or Latin1 characters.
The current implementation of Unicode / UTF-8 in FLTK will impose the following limitations:
• An implementation note in the [OksiD] code says that all functions are LIMITED to 24 bit Unicode values, but also says that only 16 bits are really used under linux and win32. [Can we verify this?]
• The [fltk2] fl_utf8encode() and fl_utf8decode() functions are designed to handle Unicode characters in the range U+000000 to U+10FFFF inclusive, which covers all UTF-16 characters, as specified in RFC 3629.

Note that the user must first convert UTF-16 surrogate pairs to UCS.

• FLTK will only handle single characters, so composed characters consisting of a base character and floating accent characters will be treated as multiple characters.
• FLTK will only compare or sort strings on a byte by byte basis and not on a general Unicode character basis.
• FLTK will not handle right-to-left or bi-directional text.
Todo Verify 16/24 bit Unicode limit for different character sets? OksiD's code appears limited to 16-bit whereas the FLTK2 code appears to handle a wider set. What about illegal characters? See comments in fl_utf8fromwc() and fl_utf8toUtf16().

Illegal Unicode and UTF-8 Sequences

Three pre-processor variables are defined in the source code [1] that determine how fl_utf8decode() handles illegal UTF-8 sequences:
• if ERRORS_TO_CP1252 is set to 1 (the default), fl_utf8decode() will assume that a byte sequence starting with a byte in the range 0x80 to 0x9f represents a Microsoft CP1252 character, and will return the value of an equivalent UCS character. Otherwise, it will be processed as an illegal byte value as described below.
• if STRICT_RFC3629 is set to 1 (not the default!) then UTF-8 sequences that correspond to illegal UCS values are treated as errors. Illegal UCS values include those above U+10FFFF, or corresponding to UTF-16 surrogate pairs. Illegal byte values are handled as described below.
• if ERRORS_TO_ISO8859_1 is set to 1 (the default), the illegal byte value is returned unchanged, otherwise 0xFFFD, the Unicode REPLACEMENT CHARACTER, is returned instead.
[1] Since FLTK 1.3.4 you may set these three pre-processor variables on your compile command line with D"variable=value" (value: 0 or 1) to avoid editing the source code.

fl_utf8encode() is less strict, and only generates the UTF-8 sequence for 0xFFFD, the Unicode REPLACEMENT CHARACTER, if it is asked to encode a UCS value above U+10FFFF.

Many of the [fltk2] functions below use fl_utf8decode() and fl_utf8encode() in their own implementation, and are therefore somewhat protected from bad UTF-8 sequences.
The [OksiD] fl_utf8len() function assumes that the byte it is passed is the first byte in a UTF-8 sequence, and returns the length of the sequence. Trailing bytes in a UTF-8 sequence will return -1.

• WARNING: fl_utf8len() can not distinguish between single bytes representing Microsoft CP1252 characters 0x80-0x9f and those forming part of a valid UTF-8 sequence. You are strongly advised not to use fl_utf8len() in your own code unless you know that the byte sequence contains only valid UTF-8 sequences.

• WARNING: Some of the [OksiD] functions below still use fl_utf8len() in their implementations. These may need further validation.
Please see the individual function description for further details about error handling and return values.

FLTK Unicode and UTF-8 Functions

This section currently provides a brief overview of the functions.

`declare function fl_utf8locale() as long` FLTK2
fl_utf8locale() returns true if the "locale" seems to indicate that UTF-8 encoding is used.
It is highly recommended that you change your system so this does return true!

`declare function fl_utf8test(src as const zstring ptr, len_ as unsigned long) as long` FLTK2
fl_utf8test() examines the first len bytes of src. It returns 0 if there are any illegal UTF-8 sequences; 1 if src contains plain ASCII or if len is zero; or 2, 3 or 4 to indicate the range of Unicode characters found.

`declare function fl_utf_nb_char(buf as const unsigned byte ptr, len_ as long) as long` OksiD
Returns the number of UTF-8 characters in the first len bytes of buf.

`declare function fl_utf8bytes(ucs as unsigned long) as long`
Returns the number of bytes needed to encode ucs in UTF-8.

`declare function fl_utf8len(c as byte) as long` OksiD
If c is a valid first byte of a UTF-8 encoded character sequence, fl_utf8len() will return the number of bytes in that sequence. It returns -1 if c is not a valid first byte.

`declare function fl_nonspacing(ucs as unsigned long) as unsigned long` OksiD
Returns true if ucs is a non-spacing character.

`declare function fl_utf8back(p as const zstring ptr, start as const zstring ptr, end_ as const zstring ptr) as const zstring ptr` FLTK2
`declare function fl_utf8fwd(p as const zstring ptr, start as const zstring ptr, end_ as const zstring ptr) as const zstring ptr` FLTK2
If p already points to the start of a UTF-8 character sequence, these functions will return p. Otherwise fl_utf8back() searches backwards from p and fl_utf8fwd() searches forwards from p, within the start and end limits, looking for the start of a UTF-8 character.

`declare function fl_utf8decode(p as const zstring ptr, end as const zstring ptr, len_ as long ptr) as unsigned long` FLTK2
`declare function fl_utf8encode(ucs as unsigned long, buf as zstring ptr) as long` FLTK2
fl_utf8decode() attempts to decode the UTF-8 character that starts at p and may not extend past end. It returns the Unicode value, and the length of the UTF-8 character sequence is returned via the len argument.
fl_utf8encode() writes the UTF-8 encoding of ucs into buf and returns the number of bytes in the sequence. See the main documentation for the treatment of illegal Unicode and UTF-8 sequences.

`declare function fl_utf8froma (dst as zstring ptr, dstlen as unsigned long, src as const zstring ptr, srclen as unsigned long) as unsigned long` FLTK2
`declare function fl_utf8toa (src as const zstring ptr, srclen as unsigned long, dst as zstring ptr, dstlen as unsigned long) as unsigned long` FLTK2
fl_utf8froma() converts a character string containing single bytes per character (i.e. ASCII or ISO-8859-1) into UTF-8. If the src string contains only ASCII characters, the return value will be the same as srclen.
fl_utf8toa() converts a string containing UTF-8 characters into single byte characters. UTF-8 characters that do not correspond to ASCII or ISO-8859-1 characters below 0xFF are replaced with '?'.
Both functions return the number of bytes that would be written, not counting the null terminator. Dstlen provides a means of limiting the number of bytes written, so setting dstlen to zero is a means of measuring how much storage would be needed before doing the real conversion.

`declare function fl_utf2mbcs(src as const zstring ptr) as zstring ptr` OksiD
converts a UTF-8 string to a local multi-byte character string. [More info required here!]

`declare function fl_utf8fromwc(dst as const zstring ptr, dstlen as unsigned long, src as const wstring ptr, srclen as unsigned long) as unsigned long` FLTK2
`declare function fl_utf8towc(src as const zstring ptr, srclen as unsigned long, dst as const wstring ptr, dstlen as unsigned long) as unsigned long` FLTK2
`declare function fl_utf8toUtf16(src as const zstring ptr, srclen as unsigned long, dst as unsigned short ptr, dstlen as unsigned long) as unsigned long` FLTK2
These routines convert between UTF-8 and wchar_t or "wide character" strings. The difficulty lies in the fact that sizeof(wchar_t) is 2 on Windows and 4 on Linux and most other systems. Therefore some "wide characters" on Windows may be represented as "surrogate pairs" of more than one wchar_t.
fl_utf8fromwc() converts from a "wide character" string to UTF-8. Note that srclen is the number of wchar_t elements in the source string and on Windows this might be larger than the number of characters.

dstlen specifies the maximum number of bytes to copy, including the null terminator.
fl_utf8towc() converts a UTF-8 string into a "wide character" string. Note that on Windows, some "wide characters" might result in "surrogate pairs" and therefore the return value might be more than the number of characters. dstlen specifies the maximum number of wchar_t elements to copy, including a zero terminating element.

fl_utf8toUtf16() converts a UTF-8 string into a "wide character" string using UTF-16 encoding to handle the "surrogate pairs" on Windows. dstlen specifies the maximum number of wchar_t elements to copy, including a zero terminating element. These routines all return the number of elements that would be required for a full conversion of the src string, including the zero terminator. Therefore setting dstlen to zero is a way of measuring how much storage would be needed before doing the real conversion.

`declare function fl_utf8from_mb(dst as zstring ptr, dstlen as unsigned long, src as const zstring ptr, srclen as unsigned long) as unsigned long` FLTK2
`declare function fl_utf8to_mb(src as const zstring ptr, srclen as unsigned long, dst as zstring ptr, dstlen as unsigned long) as unsigned long` FLTK2
These functions convert between UTF-8 and the locale-specific multi-byte encodings used on some systems for filenames, etc. If fl_utf8locale() returns true, these functions don't do anything useful.

`declare function fl_tolower(ucs as unsigned long) as long` OksiD
`declare function fl_toupper(ucs as unsigned long) as long` OksiD
`declare function fl_utf_tolower(str_ as const unsigned byte ptr, len_ as long, buf as zstring ptr) as long` OksiD
`declare function fl_utf_toupper(str_ as const unsigned byte ptr, len_ as long, buf as zstring ptr) as long` OksiD
fl_tolower() and fl_toupper() convert a single Unicode character from upper to lower case, and vice versa. fl_utf_tolower() and fl_utf_toupper() convert a string of bytes, some of which may be multi-byte UTF-8 encodings of Unicode characters, from upper to lower case, and vice versa.

Warning: to be safe, buf length must be at least 3*len [for 16-bit Unicode]

`declare function fl_utf_strcasecmp(s1 as const zstring ptr, s2 as const zstring ptr) as long` OksiD
`declare function fl_utf_strncasecmp(s1 as const zstring ptr, s2 as const zstring ptr, n as long) as long` OksiD
fl_utf_strcasecmp() is a UTF-8 aware string comparison function that converts the strings to lower case Unicode as part of the comparison. flt_utf_strncasecmp() only compares the first n characters.

FLTK Unicode Versions of System Calls

- `declare function fl_access(f as const zstring ptr, mode as long) as long` OksiD
- `declare function fl_chmod(f as const zstring ptr, mode as long) as long` OksiD
- `declare function fl_execvp(file as const zstring ptr, argv as zstring ptr const ptr) as long` OksiD
- `declare function fl_fopen(f as const zstring ptr, mode as const zstring ptr) as FILE ptr` OksiD
- `declare function fl_getcwd(buf as zstring ptr, maxlen as long) as zstring ptr` OksiD
- `declare function fl_getenv(name as const zstring ptr) as zstring ptr` OksiD
- `declare function fl_make_path( path as const zstring ptr ) as byte` OksiD
- `declare sub fl_make_path_for_file( path as const zstring ptr )` OksiD
- `declare function fl_mkdir(f as const zstring ptr, mode as long) as long` OksiD
- `declare function fl_open(f as const zstring ptr, o as long, ...) as long` OksiD
- `declare function fl_rename(f as const zstring ptr, t as const zstring ptr) as long` OksiD
- `declare function fl_rmdir(f as const zstring ptr) as long` OksiD
- `declare function fl_stat(path as const zstring ptr, buffer as any ptr ) as long` OksiD
- `declare function fl_system(f as const zstring ptr) as long` OksiD
- `declare function fl_unlink(f as const zstring ptr) as long` OksiD

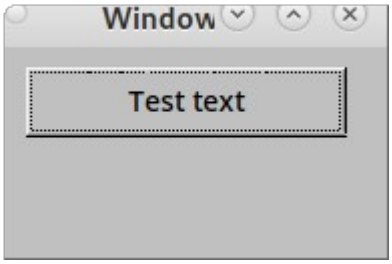
Examples

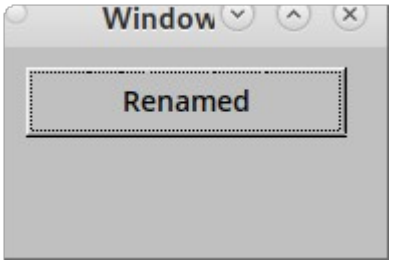
Project 006-callbacks01

```
1.  #include once "FLTK/Fl_Double_Window.bi"
2.  #include once "FLTK/Fl_Button.bi"
3.
4.  sub rename_me cdecl(o as Fl_Widget ptr, p as any ptr)
5.        o->label("Renamed")
6.        o->redraw()
7.  end sub
8.
9.  '' main program
10. Dim w as Fl_Double_Window = Fl_Double_Window(200, 105, "Window")
11.       Dim b1 as Fl_button = Fl_button(20, 10, 160, 35, "Test text")
12.       Dim cb as Fl_Callback = @rename_me
13.       b1.callback(cb)
14. w.end_()
15. w.resizable(@b1)
16. w.show()
17.
18. fl.run_
```

Analysis
In this Project we want to press a button and change the text of the button's label.
We include Fl_Double_Window and Fl_Button at lines 1-2
At lines 4-7 exists the callback function for renaming the label of button.
The sub rename_me accepts as argument a Fl_Widget pointer
Next, at line 5 we set the new label text and finally at line 6 we redraw the widget
At lines 9-18 exists the main program.
We create the Double Window (line 10) and a Button (line 11), next we asssign the button's callback (lines 12-13).
Next we end the window scope (line 14) and set button to be resizable according with the window (line 15).
Finally we show the window (line 16) and enter the FLTK event loop (line 18).

Output
The running application is shown at the following images





A resizable window is shown with a button.
When user clicks the button, it changes the text of its label.

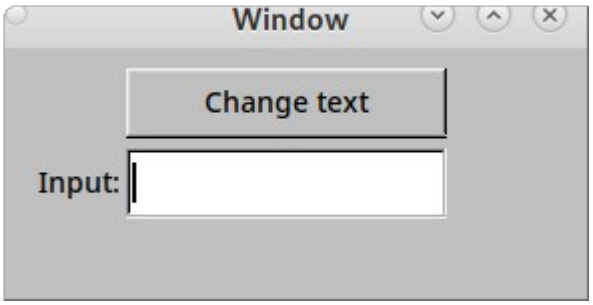
Project 007-input

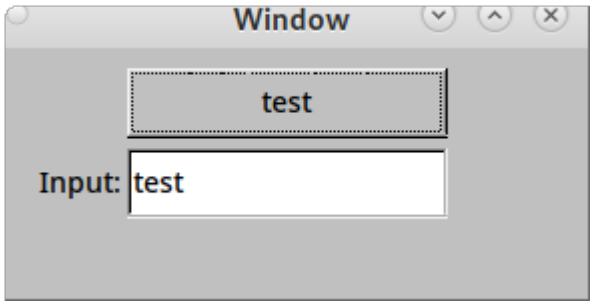
On this application we have a resizable window, a button and a input box (textbox).
We want to change button's label according to the input of textbox.

```
1.  #include once "FLTK/Fl_Double_Window.bi"
2.  #include once "FLTK/Fl_Button.bi"
3.  #include once "FLTK/Fl_Input.bi"
4.
5.  sub rename_me(o as Fl_Widget ptr, s as any ptr)
6.        dim s0 as Fl_Input ptr = s
7.        o->label(s0->value)
8.        o->redraw()
9.  end sub
10.
11. '' main program
12. Dim w as Fl_Double_Window = Fl_Double_Window(300, 125, "Window")
13.       Dim sInput1 As Fl_Input = Fl_Input(70,50,160,35,"Input:")
14.       Dim b1 as Fl_button = Fl_button(70, 10, 160, 35, "Change text")
15.       Dim cb as Fl_Callback = @rename_me
16.       b1.callback(cb, @sInput1)
17.
18. w.end_()
19. w.resizable(@b1)
20. w.resizable(@sInput1)
21. w.show()
22.
23. fl.run_
```

Analysis
We include Fl_Double_Window, Fl_Button and Fl_Input at lines 1-3
At lines 5-9 exists the callback function for renaming the label of button.
The sub rename_me accepts as argument a Fl_Widget object and a pointer.
At line 6 we define a Fl_Input pointer and assign to it the s pointer.
Next, at line 7 we set the new label text and finally at line 8 we redraw the widget
At lines 11-23 exists the main program.
We create the Double Window (line 12), an Input textbox (line 13) and a Button (line 14), next we asssign the button's callback (lines 15-16).
Next we end the window scope (line 18) and set the button and the input to be resizable according with the window (lines 19-20).
Finally we show the window (line 16) and enter the FLTK event loop (line 23).

Output





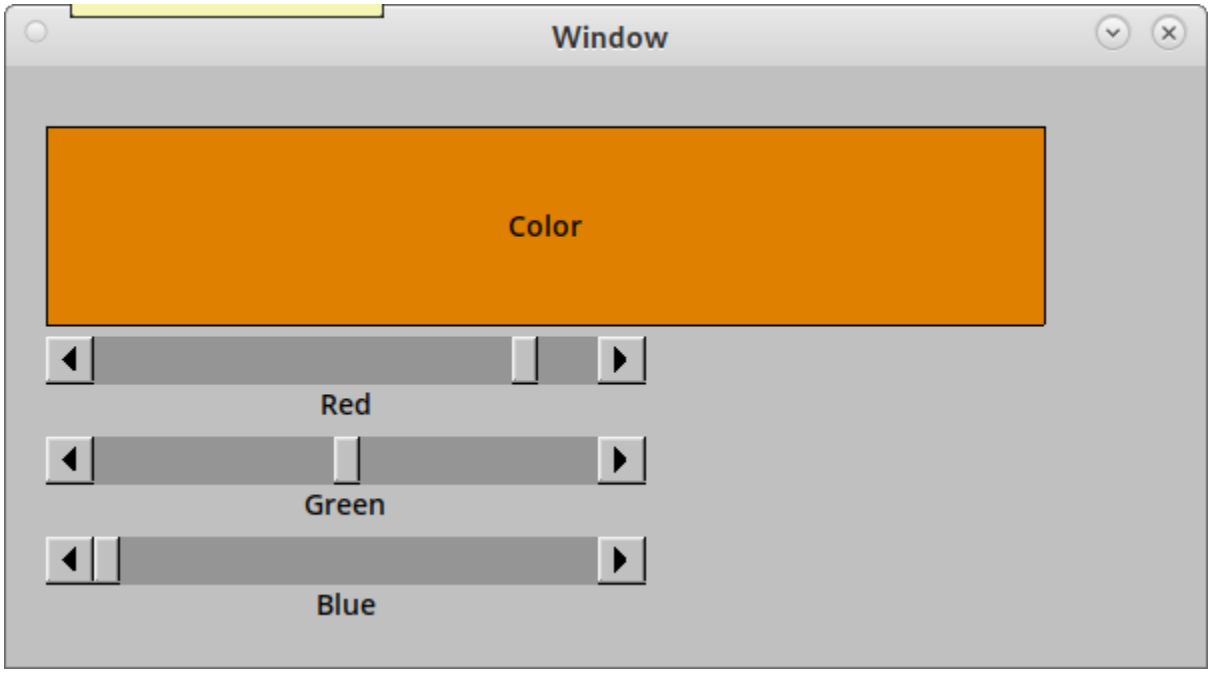
Project 008-colorobox

On this application we have a window, a Box and three horizontal scrollbars..
We want to change the color of the Box according to Red, Green, Blue values of scrollbars

```
#include once "FLTK/Fl_Double_Window.bi"
#include once "FLTK/Fl_Scrollbar.bi"
#include once "FLTK/Fl_Box.bi"

declare sub setColor(as Fl_Widget ptr, as any ptr)

'' main program
Dim shared w as Fl_Double_Window = _
      Fl_Double_Window(600, 300, "Window")
      Dim shared box As Fl_Box =_
            Fl_Box(FL_BORDER_BOX,20,30,500,100,"Color")
      Dim shared hScrollBarRed As Fl_Scrollbar=_
            Fl_Scrollbar(20,135,300,24,"Red")
      Dim shared hScrollBarGreen As Fl_Scrollbar=_
            Fl_Scrollbar(20,185,300,24,"Green")
      Dim shared hScrollBarBlue As Fl_Scrollbar=_
            Fl_Scrollbar(20,235,300,24,"Blue")

      hScrollBarRed.type_(FL_HORIZONTAL)
      hScrollBarGreen.type_(FL_HORIZONTAL)
      hScrollBarBlue.type_(FL_HORIZONTAL)

      hScrollBarRed.minimum(0)
      hScrollBarRed.maximum(255)
      hScrollBarRed.value(0)

      hScrollBarGreen.minimum(0)
      hScrollBarGreen.maximum(255)
      hScrollBarGreen.value(128)

      hScrollBarBlue.minimum(0)
      hScrollBarBlue.maximum(255)
      hScrollBarBlue.value(0)

      Dim c as Fl_Color = fl_rgb_color(hScrollBarRed.value,_
                                       hScrollBarGreen.value,_
                                       hScrollBarBlue.value)
      box.Color(c)

      Dim cb as Fl_Callback = @setColor
      hScrollBarRed.callback(cb)
      hScrollBarGreen.callback(cb)
      hScrollBarBlue.callback(cb)

w.end_()
w.show()

fl.run_
'' end of main program

sub setColor(o as Fl_Widget ptr, p as any ptr)
      Dim c as Fl_Color = fl_rgb_color(hScrollBarRed.value,_
                                       hScrollBarGreen.value,_
                                       hScrollBarBlue.value)
      box.Color(c)
      box.redraw()
end sub
```

Analysis
We include Fl_Double_Window, Fl_Scrollbar and Fl_Box at lines 1-3
At line 5 we declare a callback sub for setting the color of box.
At lines 8-48 exists the main program of our application.
At lines 8-33 we setup the window, scrollbars and the box.
At lines 35-38 we set the color of the box for the first time.
At lines 40-43 we setup the callback sub to set the color while we click or scroll the scrollbars.
At line 45 we end the window scope.
At line 46 we show the window.
At line 48 we enter the FLTK event loop.
At lines 51-57 exists the setColor callback sub.
At lines 52-54 we set a color according the values of scrollbars.
At line 55 we set the color of the box.
At line 56 we redraw the box.

Output



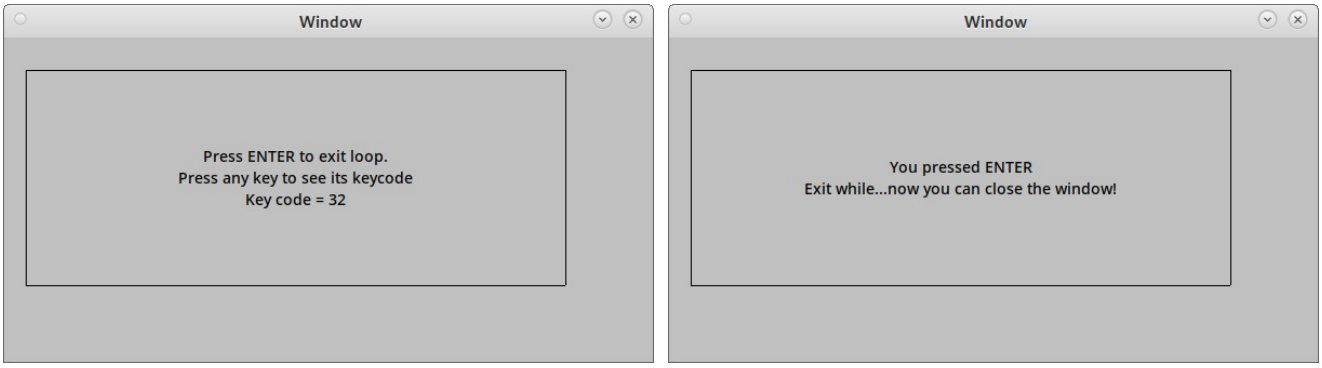
Project 009-keyboard

On this application we have a window, and a Box..
We catch keyboard pressed buttons and show the keycode in a message in the box.

```
1.  #include once "FLTK/Fl_Double_Window.bi"
2.  #include once "FLTK/Fl_Box.bi"
3.
4.  '' main program
5.  Dim z as zstring*128
6.  Dim s as string
7.  Dim w as Fl_Double_Window = _
8.        Fl_Double_Window(600, 300, "Window")
9.        Dim box As Fl_Box =_
10.             Fl_Box(FL_BORDER_BOX,20,30,500,200,"Press any key...")
11.
12.       while(true) 'catch events
13.             s=str(fl.event_key)
14.             z=!"Press ENTER to exit loop.\n"+_
15.                   !"Press any key to see its keycode\n"+_
16.                   "Key code = "+s
17.             box.label(z)
18.             box.redraw()
19.             w.show()
20.             'if user press ENTER exit loop
21.             if fl.event_key = 65293 then exit while end if
22.             fl.check()
23.       wend
24.
25.       box.label(!"You pressed ENTER\nExit while..."+_
26.                         "now you can close the window!")
27.       w.end_
28.       fl.run_
```

Analysis
At lines 1-2 we include the Double Window and Box FLTK headers.
At line 5 we define a zstring variable. The text of z variable will be set it as box's label value.
At line 6 we define a string variable. This variable is going to hold the pressed key's keycode.
At lines 7-10 we create the window and the box widgets of our application.
Next at lines 12-23 we have a while...wend loop. The loop is endless and run continuously.
In each run of the loop we catch fl.event_key which this means that we catch keyboard's pressed buttons by the user.
The variable s holds as string the keycode which is converted to string from numeric representation.
At lines 14-16 we create a string message, with the keycode as string.
At line 17 we set the box's label with the z value.
At line 18 we redraw the box widget in order to update the message in it.
At line 19 we show the window. Note that w.show() is inside while...wend loop. That means that if the user tries to close the window it will be shown again. In order to close the window the user must press ENTER key (keycode=65293) to exit loop. This is tested at line 21.
At line 22 we call fl.check() which keeps the screen up to date and the interface responsive.
At line 25 we are outside the while...wend loop and show a message for it in the box's label.

Output



At left we have pressed the SPACEBAR key from keyboard, keycode=32.
And at the right we have pressed the ENTER key from keyboard.

Project 010-drag-n-drop

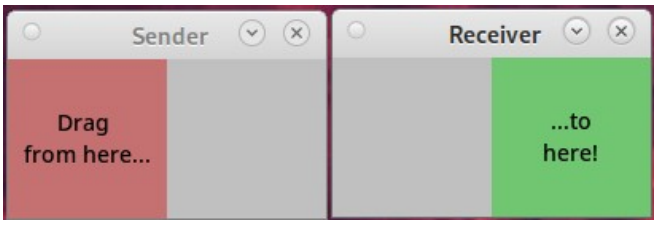
On this application we have two windows, with a Box on each.
We drag from first window's box and drop to second window's box.
If drag n drop is successful the text "It works" appears in second's window box.

```
#include once "FLTK/Fl_Window.bi"
#include once "FLTK/Fl_Box.bi"

'SIMPLE SENDER UDT
Type sender extends Fl_Box
      declare constructor()
      declare constructor (byref w as const sender)
      declare operator let (byref w as const sender)
      declare function handle(as long) as long
End Type

constructor sender()
      base(FL_FLAT_BOX, 0,0,100,100,"Sender")
end constructor

function sender.handle(event as long) as long
      dim ret as long = base.handle(event)
      select case event
            case FL_PUSH
                  dim msg as string = "It works!"
                  Fl.copy(msg, len(msg),0)
                  Fl.dnd()
                  ret=1
      end select
      return(ret)
end function

'SIMPLE RECIEVER UDT
Type reciever extends Fl_Box
      declare constructor()
      declare constructor (byref w as const reciever)
      declare operator let (byref w as const reciever)
      declare function handle(as long) as long
End Type

constructor reciever()
      base(FL_FLAT_BOX, 100,0,100,100,"Reciever")
```

```
38. end constructor
39.
40. function reciever.handle(event as long) as long
41.       dim ret as long = base.handle(event)
42.       select case event
43.             case FL_DND_ENTER 'return(1) for these events to 'accept' dnd
44.                   ret=1
45.             case FL_DND_DRAG
46.                   ret=1
47.             case FL_DND_RELEASE
48.                   ret=1
49.             case FL_PASTE     'handle actual drop (paste) operation
50.                   this.label(Fl.event_text())
51.                   ret=1
52.       end select
53.       return(ret)
54. end function
55.
56. ''main program
57. 'Create sender window and widget
58. Dim win_a as Fl_Window = Fl_Window(0,0,200,100,"Sender")
59.       Dim a as sender = sender()
60.       a.Color(9)
61.       a.label(!"Drag \nfrom here...")
62. win_a.end_()
63. win_a.show()
64.
65. 'Create receiver window and widget
66. Dim win_b as Fl_Window = Fl_Window(400,0,200,100,"Receiver")
67.       Dim b as reciever= reciever()
68.       b.Color(10)
69.       b.label(!"...to\nhere!")
70. win_b.end_()
71. win_b.show()
72.
73. Fl.run_()
```

Output

Before dnd



After dnd

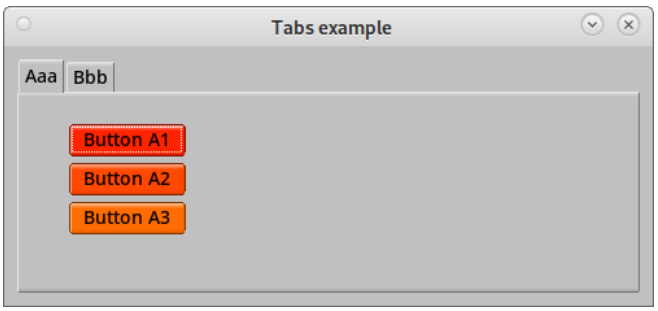
Project 011-tabs-simple

On this application we have one window, with two tabs.
Each tab has a group of buttons

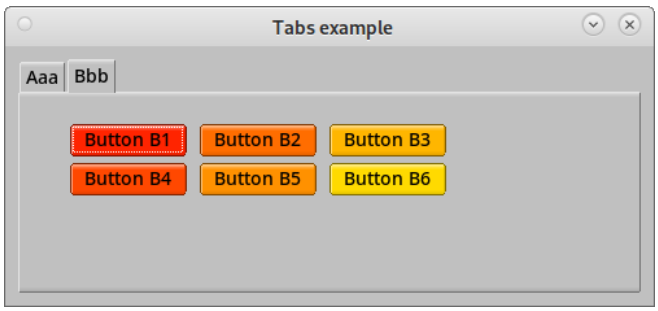
```
#include once "FLTK/Fl_Window.bi"
#include once "FLTK/Fl_Tabs.bi"
#include once "FLTK/Fl_Group.bi"
#include once "FLTK/Fl_Button.bi"

' Simple tabs example
'       _____  _____
'    __/ Aaa \/ Bbb \_____________________
'   |     ________                        |
'   |    |________|                       |
'   |     ________                        |
'   |    |________|                       |
'   |     ________                        |
'   |    |________|                       |
'   |_____________________________________|
'
Fl.scheme("gtk+")
Dim win as Fl_Window = Fl_Window(0,0,500,200,"Tabs example")
  Dim tabs as Fl_Tabs = Fl_Tabs(10,10,500-20,200-20)
    Dim aaa as Fl_Group = Fl_Group(10,35,500-20,200-45,"Aaa")
      Dim b1 as Fl_Button = Fl_Button(50, 60,90,25,"Button A1")
      b1.color(88+1)
      Dim b2 as Fl_Button = Fl_Button(50, 90,90,25,"Button A2")
      b2.color(88+2)
      Dim b3 as Fl_Button = Fl_Button(50,120,90,25,"Button A3")
      b3.color(88+3)
    aaa.end_()

    Dim bbb as Fl_Group = Fl_Group(10,35,500-10,200-35,"Bbb")
      Dim b4 as Fl_Button = Fl_Button( 50,60,90,25,"Button B1")
      b4.color(88+1)
      Dim b5 as Fl_Button = Fl_Button(150,60,90,25,"Button B2")
      b5.color(88+3)
      Dim b6 as Fl_Button = Fl_Button(250,60,90,25,"Button B3")
      b6.color(88+5)
      Dim b7 as Fl_Button = Fl_Button( 50,90,90,25,"Button B4")
      b7.color(88+2)
      Dim b8 as Fl_Button = Fl_Button(150,90,90,25,"Button B5")
      b8.color(88+4)
      Dim b9 as Fl_Button = Fl_Button(250,90,90,25,"Button B6")
      b9.color(88+6)
     bbb.end_()
    tabs.end_()
win.end_()
win.show()
Fl.run_()
```

Output

Tab1



Tab2



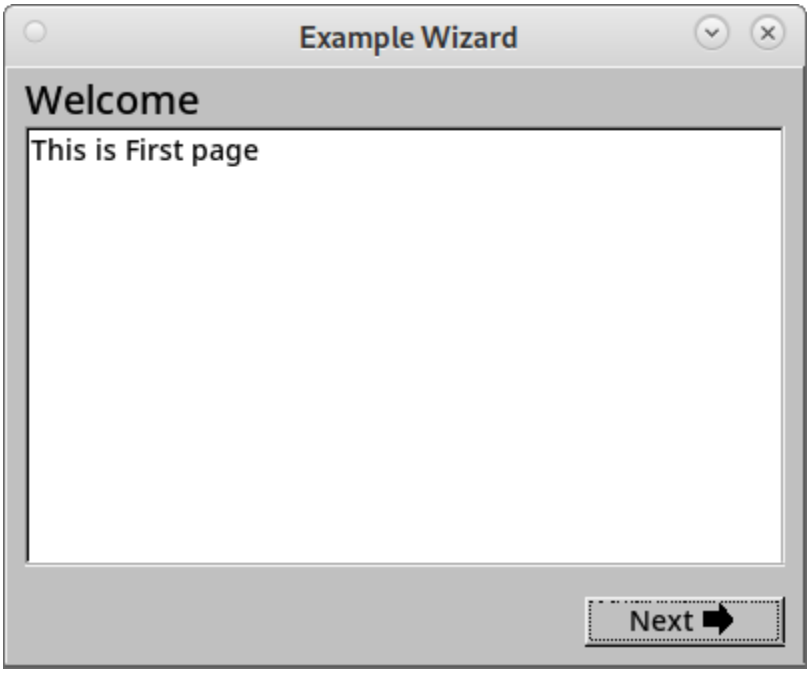
Project 012-wizard-simple

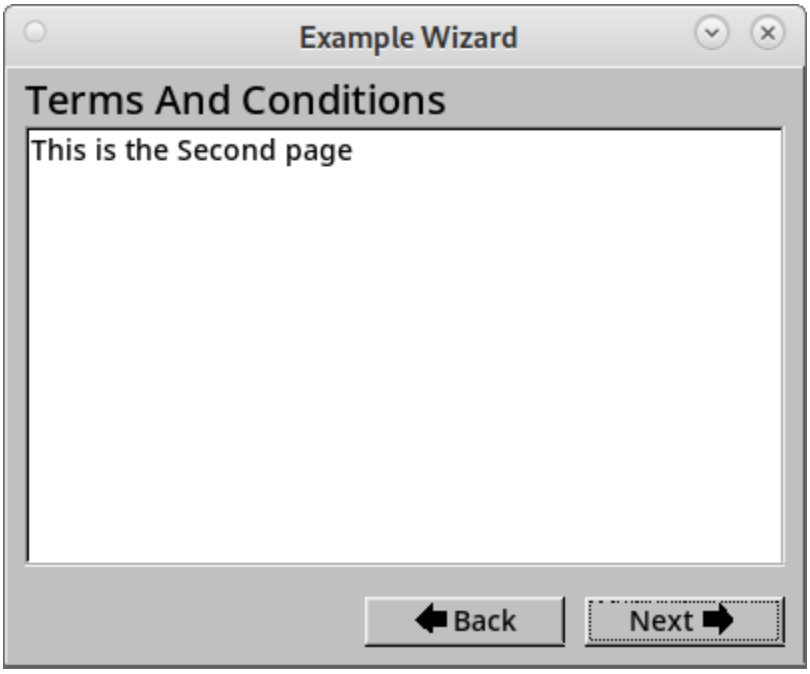
On this application we have one window, with a wizard widget.

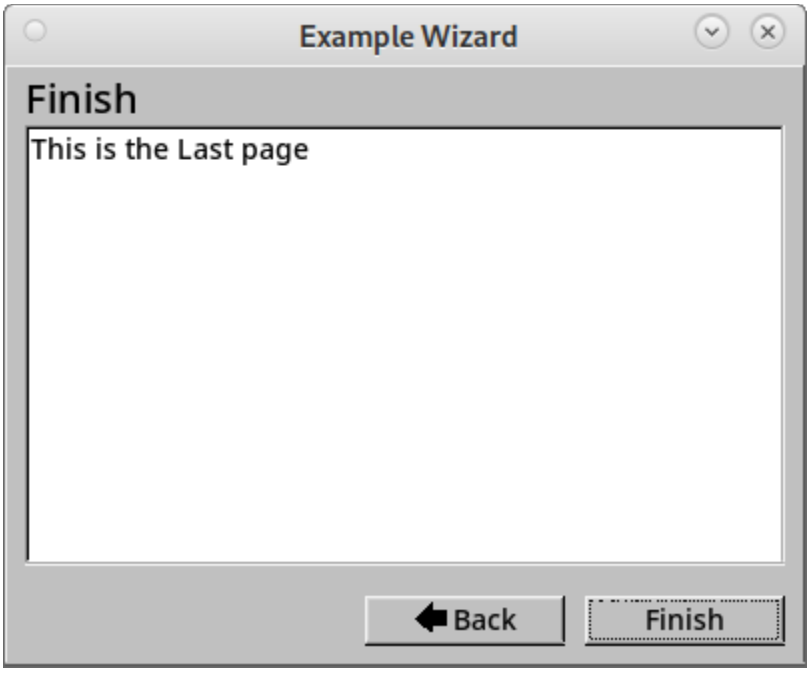
```
1.  #include once "FLTK/Fl_Window.bi"
2.  #include once "FLTK/Fl_Group.bi"
3.  #include once "FLTK/Fl_Wizard.bi"
4.  #include once "FLTK/Fl_Button.bi"
5.  #include once "FLTK/Fl_Multiline_Output.bi"
6.
7.  Dim shared G_Win as Fl_Window = Fl_Window(400,300,"Example Wizard")
8.  Dim shared G_Wiz as Fl_Wizard = Fl_Wizard(0,0,400,300)
9.
10. public sub back_cb cdecl(o as Fl_Widget ptr, p as any ptr)
11.       G_Wiz.prev()
12. end sub
13.
14. public sub next_cb cdecl(o as Fl_Widget ptr, p as any ptr)
15.       G_Wiz.next_()
16. end sub
17.
18. public sub done_cb cdecl(o as Fl_Widget ptr, p as any ptr)
19.       end
20. end sub
21.
22. ' Wizard: page 1
23. Dim g1 as Fl_Group = Fl_Group(0,0,400,300)
24. Dim next1 as Fl_Button = Fl_Button(290,265,100,25,"Next @->")
25. Dim cb1 as Fl_Callback = @next_cb
26. next1.callback(cb1)
27. Dim out1 as Fl_Multiline_Output = Fl_Multiline_Output(10,30,400-20,300-
    80,"Welcome")
28. out1.labelsize(20)
29. out1.align(FL_ALIGN_TOP or FL_ALIGN_LEFT)
30. out1.value("This is First page")
31. g1.end_()
32.
33. ' Wizard: page 2
34. Dim g2 as Fl_Group = Fl_Group(0,0,400,300)
35. Dim next2 as Fl_Button = Fl_Button(290,265,100,25,"Next @->")
36. next2.callback(cb1)
37. Dim back1 as Fl_Button = Fl_Button(180,265,100,25,"@<- Back")
38. Dim cb2 as Fl_Callback = @back_cb
39. back1.callback(cb2)
40. Dim out2 as Fl_Multiline_Output = Fl_Multiline_Output(10,30,400-20,300-
```

```
    80,"Terms And Conditions")
41. out2.labelsize(20)
42. out2.align(FL_ALIGN_TOP or FL_ALIGN_LEFT)
43. out2.value("This is the Second page")
44. g2.end_()
45.
46. ' Wizard: page 3
47. Dim g3 as Fl_Group = Fl_Group(0,0,400,300)
48. Dim  done as Fl_Button = Fl_Button(290,265,100,25,"Finish")
49. Dim cb3 as Fl_Callback = @done_cb
50. done.callback(cb3)
51. Dim back as Fl_Button = Fl_Button(180,265,100,25,"@<- Back")
52. back.callback(cb2)
53. Dim out3 as Fl_Multiline_Output = Fl_Multiline_Output(10,30,400-20,300-
    80,"Finish")
54. out3.labelsize(20)
55. out3.align(FL_ALIGN_TOP or FL_ALIGN_LEFT)
56. out3.value("This is the Last page")
57. g3.end_()
58.
59. G_Wiz.end_()
60. G_Win.end_()
61. G_Win.show()
62. Fl.run_()
```

Output




Example Wizard
Terms And Conditions
This is the Second page
Back
Next


Example Wizard
Finish
This is the Last page
Back
Finish

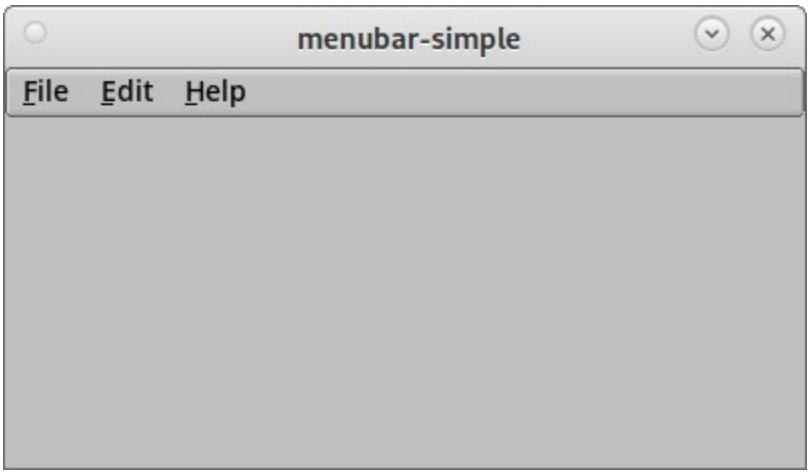
Project 013-menubar-add

On this application we have one window, with a menu bar with items.

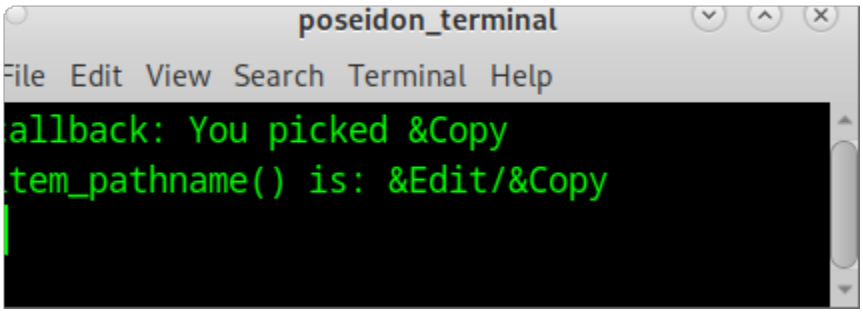
```
1.  #include once "FLTK/Fl_Window.bi"
2.  #include once "FLTK/Fl_Menu_Bar.bi"
3.  #include once "FLTK/filename.bi"
4.
5.  ' This callback is invoked whenever the user clicks an item in the menu
6.  bar
7.  Sub MyMenuCallback(w as Fl_Widget ptr, p as any ptr)
8.    dim bar as Fl_Menu_Bar ptr = cptr(Fl_Menu_Bar ptr, w)
9.    dim item as const Fl_Menu_Item ptr = cptr(Fl_Menu_Item ptr, bar-
    mvalue())
10.
11.   dim ipath as string*256
12. ' Get full pathname of picked item
13.   bar->item_pathname(ipath, sizeof(ipath))
14.
15.       print "callback: You picked " & *item->label() 'Print item picked
16.       print "item_pathname() is: " & ipath      'and full pathname
17.
18.       'Toggle or radio item?
19.        if ( item->flags and (FL_MENU_RADIO or FL_MENU_TOGGLE) ) then
20.            'Print item's value
21.             print "value is " & iif(item->value(), "on", "off")
22.       end if
23.
24.       if (*item->label() = "Google" ) then
25.             'fl_open_uri("http://google.com/")
26.       end if
27.
28.       if (*item->label()="&Quit") then
29.            end
30.       end if
31. end sub
32.
33. 'main program
34. Fl.scheme("gtk+")
35. ' Create window
36. dim win as Fl_Window = Fl_Window(400,200, "menubar-simple")
37. ' Create menubar, items
38. dim menu as Fl_Menu_Bar = Fl_Menu_Bar(0,0,400,25)
39. Dim cb as Fl_Callback = @MyMenuCallback
40. menu.add("&File/&Open", "^o", cb)
41. menu.add("&File/&Save", "^s", cb, 0, FL_MENU_DIVIDER)
42. menu.add("&File/&Quit", "^q", cb)
43. menu.add("&Edit/&Copy", "^c", cb)
44. menu.add("&Edit/&Paste", "^v", cb, 0, FL_MENU_DIVIDER)
45. menu.add("&Edit/Radio 1", 0, cb, 0, FL_MENU_RADIO)
46. menu.add("&Edit/Radio 2", 0, cb, 0, FL_MENU_RADIO or FL_MENU_DIVIDER)
47. menu.add("&Edit/Toggle 1", 0, cb, 0, FL_MENU_TOGGLE) ' Default: off
48. menu.add("&Edit/Toggle 2", 0, cb, 0, FL_MENU_TOGGLE) ' Default: off
49. ' Default: on
50. menu.add("&Edit/Toggle 3", 0, cb, 0, FL_MENU_TOGGLE or FL_MENU_VALUE)
51. menu.add("&Help/Google", 0, cb)
52.
53. win.end_()
54. win.show()
55. Fl.run_()
```

Note:
At line 25 `fl_open_uri()` is missing from the FLTK headers for now. You can uncomment the line in future releases of FLTK headers when this function will be added.

Output



We selected Edit→Copy

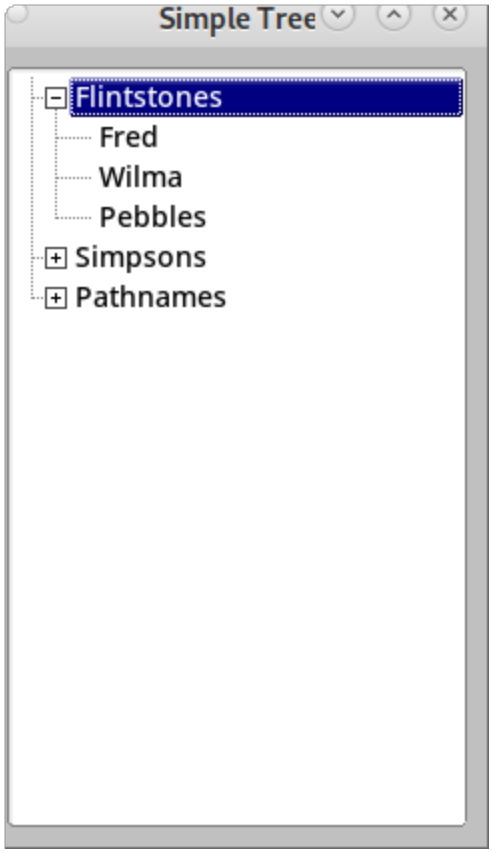


Project 014-tree-simple

On this application we have one window, with a tree widget with items.

```
#include once "FLTK/Fl_Double_Window.bi"
#include once "FLTK/Fl_Menu_Bar.bi"
#include once "FLTK/Fl_Tree.bi"

'Tree's callback
'Invoked whenever an item's state changes.
'This callback is invoked whenever the user clicks an item in the menu bar
Sub TreeCallback(w as Fl_Widget ptr, p as any ptr)
      var tree = cptr(Fl_Tree ptr, w)
  dim item as const Fl_Tree_Item ptr = cptr(Fl_Tree_Item ptr, tree-
callback_item())

      if (item <> 1) then
            exit sub
      end if

      select case (tree->callback_reason())
      case FL_TREE_REASON_SELECTED
            dim pathname as string*256
            tree->item_pathname(pathname, sizeof(pathname), item)
            print "TreeCallback: Item selected=" & item->label() & " Full
pathname= " & pathname
      case FL_TREE_REASON_DESELECTED
            print "TreeCallback: Item " & item->label() & " deselected"
      case FL_TREE_REASON_OPENED
            print "TreeCallback: Item " & item->label() & " opened"
      case FL_TREE_REASON_CLOSED
            print "TreeCallback: Item " & item->label() & " closed"
      end select
end sub

'main program
Fl.scheme("gtk+")
dim win as Fl_Double_Window = Fl_Double_Window(250, 400, "Simple Tree")
win.begin()
'Create the tree
dim tree as Fl_Tree= Fl_Tree(10, 10, win.w()-20, win.h()-20)
tree.showroot(0)  'don't show root of tree
dim cb as Fl_Callback = @TreeCallback
tree.callback(cb) 'setup a callback for the tree

'Add some items
tree.add("Flintstones/Fred")
tree.add("Flintstones/Wilma")
tree.add("Flintstones/Pebbles")
tree.add("Simpsons/Homer")
tree.add("Simpsons/Marge")
tree.add("Simpsons/Bart")
tree.add("Simpsons/Lisa")
tree.add(!"Pathnames/\\/bin") 'front slashes
tree.add(!"Pathnames/\\/usr\\/sbin")
tree.add(!"Pathnames/C:\\\\Program Files")     'backslashes
tree.add(!"Pathnames/C:\\\\Documents and Settings")

tree.close_("Simpsons")
tree.close_("Pathnames")

win.end_()
win.resizable(win)
win.show()
Fl.run_()
```

Output



Examples on the internet

The FLTK headers for FreeBASIC is an ongoing project.
You can find out more info and examples on the internet at the following address:
https://www.freebasic.net/forum/viewtopic.php?t=31809